Automated Software Synthesis for Streaming Applications on Embedded Manycore Processors

By

MATIN HASHEMI

B.S. (Sharif University of Technology) 2005
M.S. (University of California, Davis) 2008

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL AND COMPUTER ENGINEERING

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Soheil Ghiasi, Chair

Venkatesh Akella

Bevan Baas

Committee in Charge

2011

Abstract

Stream applications are characterized by the requirement to process a virtually infinite sequence of data items. They appear in many areas including communication, networking, multimedia and cryptography. Embedded manycore systems, currently in the range of hundreds of cores, have shown a tremendous potential in achieving high throughput and low power consumption for such applications.

The focus of this dissertation is on automated synthesis of parallel software for stream applications on embedded manycore systems. Automated software synthesis significantly reduces the development and debug time. The vision is to enable seamless and efficient transformation from a higher-order specification of the stream application (e.g., dataflow graph) to parallel software code (e.g., multiple `.C` files) for a given target manycore system. This automated process involves many steps that are being actively researched, including workload estimation of tasks (actors) in the dataflow graph, allocation of tasks to processors, scheduling of tasks for execution on the processors, binding of processors to physical cores on the chip, binding of communications to physical channels on the chip, generation of the parallel software code, backend code optimization and estimation of throughput.

This dissertation improves on the state-of-the-art by making the following contributions. First, a versatile task allocation algorithm for pipelined execution is proposed that is provably-efficient and can be configured to target platforms with different underlying architectures. Second, a throughput estimation method is introduced that has acceptable accuracy, high scalability with respect to the number of cores, and a high degree of freedom in targeting platforms with different underlying onchip networks. Third, a task scheduling algorithm is proposed, based on iteration overlapping techniques, which explores the trade-off between throughput and memory requirements for manycore platforms with and without FIFO-based onchip communication channels. Finally, to increase the scalability of application throughput with respect to the number of cores, a malleable dataflow specification model is proposed.

Acknowledgments

I would like to express deep appreciation for my current and previous teachers, mentors and advisers whom I learned a lot form. In specific, I would like to wholeheartedly thank Professor Soheil Ghiasi, my PhD adviser, for his help, support and guidance during the period of my graduate studies at UC Davis. Many thanks to UC Davis faculty for teaching informative courses, and specially Professor Venkatesh Akella and Professor Bevan Baas who also improved this work with their thoughtful comments. I would like to thank the ECE department staff specially Renee Kuehnau and Lance Halsted for their passionate work and supporting of students. The credit for this work partially goes to Po-Kuan Huang, Adam Harbour, Mohammad Hossein Foroozannejad, Lin Chen, Christoph Etzel, Houshmand Shiranimehr, Bin Liu, Mehrdad Afshari and Volodymyr Khibin.

Contents

CHAPTER 1

Introduction

Streaming applications are characterized by the requirement to process a virtually infinite sequence of data items [**LP95**]. The main quality metric in the streaming domain is throughput, the rate at which data items are processed [**GGS$^+$06**]. Such applications are becoming increasingly important and widespread. They appear in many areas ranging from communications in embedded sensor nodes, to multimedia and networking in desktops computers, to high-end server applications such as cryptography, hyperspectral imaging and cellular base stations [**dK02, MKWS07, TA10**]. Because of abundant parallelism and predictable data accesses, they are capable of being efficiently executed on manycore systems [**GTA06, AED07**].

Manycore systems consist of a large number of processor cores interconnected together and behaving as a massively parallel computer system [**ABC$^+$06, Bor07**]. Performance is envisioned to scale mainly through systemwide coarse-grain parallelism. Complicated control and data dependency resolving hardware are likely to be removed in order to open more room for larger number of cores [**BHH$^+$07, AL07**]. While mainstream general purpose processors such as Intel Xeon [**SDS$^+$11**] are currently in the range of ten cores, embedded manycore systems such as Cisco CRS [**Eat05**], PicoArray [**PTD$^+$06**], TILE64 [**BEA$^+$08**] and AsAP [**TCM$^+$09**] are already in the range of hundreds of cores (Figure 1.1).

Manycore systems have shown a tremendous potential in achieving high throughput and low power consumption for embedded streaming applications [**XB08, Pul08, TB10**]. Despite this potential, however, the process of stream application development, and embedded application development in general, is largely ad-hoc today [**SGM$^+$05, Lee06, HS06**]. Presently, the practitioners have to settle for slow and costly development procedures, which are error-prone and yield unportable software [**Lee05**]. Many researchers are working to develop a formal science to pave the way to systematic and high-confidence development of

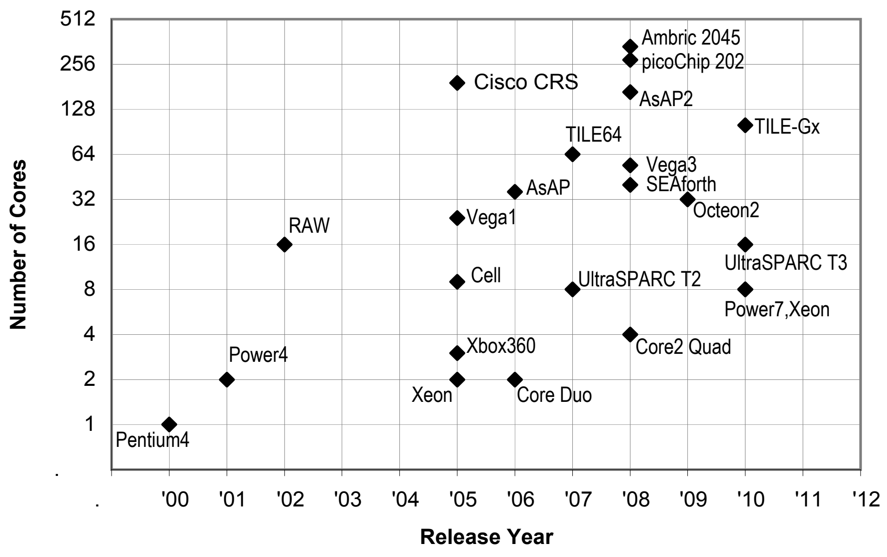


Figure 1.1. Current trend in the microprocessor design is to increase the number of cores.

embedded applications [**SGM+05, Mee06**]. As a step towards productive stream application development, we focus on automated software synthesis from high-level specifications. Automated software synthesis significantly reduces the development and debug time. The vision is to enable seamless and efficient transformation from a higher-order specification of the stream application to parallel software code (e.g., multiple `.C` files) for a given target manycore system.

Thread-based specifications often lead to unreliable software [**SL05, Lee06**], mainly because concurrency constructs in Von Neumann languages introduce nondeterminism into programs [**Wen75**]. Resolving issues, such as deadlocks, in a flawed multi-threaded program is time-consuming and often requires runtime monitoring [**JPSN09, BYLN09**]. Dataflow-based high-level specifications, however, address the scalability and productivity issues of parallel threads [**TKA02, Lee06**]. In this model, the streaming application is described as a set of explicitly-parallel communicating tasks (actors). The rate of communications among the tasks are often known or predictable in the streaming domain [**LM87a, TKA02, TKS+05**]. Hence, dataflow specifications provide a solid ground for reliable and productive application development through compile-time analysis and optimization algorithms.

In recent years, a number of research groups have worked on automated stream software synthesis from dataflow models to parallel platforms [**GTA06, SGB08, HCK+09**]. Languages such as StreamIt [**TKA02**] have been developed specifically for this purpose.

In order to automatically realize an application software, a sequence of refinement stages have to be carried out to bridge the gap between high-level dataflow specifications and application implementation. Examples include allocation and scheduling of the tasks on the processors. Chapter 2 presents an overview of different steps normally involved in stream software synthesis from dataflow models to manycore platforms.

The major contribution of this dissertation is presented in Chapters 3 to 6. Chapter 3 presents a versatile dual-processor task assignment algorithm that is provably-efficient and can be easily configured to target platforms with different underlying architectures, and also, a heuristic task assignment method for pipelined architectures. Chapter 4 presents a manycore performance estimation method with acceptable accuracy, high scalability with respect to the number of processors, and high degree of freedom in targeting platforms with different underlying onchip network architectures. Chapter 5 presents iteration overlapping which is a technique in adjusting the execution order of tasks, and is closely related to software pipelining. It greatly improves the scalability of throughput with respect to the number of processors by providing an adjustable knob in trading memory for more throughput. Chapter 6 presents a new dataflow model for specifying stream applications. Gaining insight from our previous works, we concluded that in addition to efficient optimization algorithms, *malleable* high-level specification models are required in order to achieve high scalability and portability with respect to the number of processors across a wide range of platforms. And finally Chapter 7 concludes this dissertation by presenting an overall summary of the lessons learned and possible directions for future work.

CHAPTER 2

Background

Right abstractions are the key in quality software synthesis in the embedded domain [**PBSV+06**]. Sections 2.1 and 2.2 present an overview of the abstract models that we used for the manycore platform and the streaming application specification. Note that in our empirical evaluations, the automatically synthesized codes are compiled and executed on the actual target hardware platform. Section 2.3 presents the optimization steps involved in automated software synthesis from dataflow models to manycore platforms.

2.1. Manycore Systems

A traditional parallel processing platform is composed of memory units, processor chips and their interconnection network. A manycore system, however, is fabricated entirely in one chip with the exception of memory which may or may not be integrated on the chip. Similar to the traditional platforms, manycore systems can be classified as shared-memory or distributed-memory. All processor cores have access to the same memory in a shared-memory manycore system, while in a distributed-memory system, the cores can only access their own memory.

The processor cores may cooperate in processing the same data, in which case, locks or similar mechanisms are required for synchronization. This approach is common in thread-based programming (see Section 2.2). Alternatively, the cores may communicate by sending and receiving messages, which is known as the message passing approach.

We target execution platforms whose abstract model exposed to the software synthesis process can be viewed as a distributed-memory manycore system in which the processor cores communicate by sending and receiving messages through point-to-point unidirectional FIFO channels. Figure 2.2.A on page 8 illustrates an example system with 4 processor cores in which neighbor cores are connected through FIFO channels. Many existing manycores such as AsAP [**TCM+08**] and Tilera [**BEA+08**] conform to this abstraction. FPGA-based systems can also implement the above model through soft processor cores such as Microblaze

[**Mic**] or Nios [**Nio**]. The model is reasonably accurate at high-level for other platforms that implement the abstract view using different underlying architectures.

Systems with shared memory, such as multicore Intel Xeon [**RTM$^+$09**], can be programmed to implement the virtual directed links, for example, with a virtual FIFO as an array in the shared memory space. Systems with on-chip networks, such as IBM Cell [**KDH$^+$05**], can implement the virtual directed links with assistance of system software for buffering and reordering of packets at processors. Note that the implemented communication mechanism has to deliver messages in the order they are sent. Message delivery in deterministic order, e.g. in-order delivery, is required for correct realization of statically scheduled applications.

Our model is a multiple-instruction multiple-data (MIMD) architecture which is conceptually different from a single-instruction multiple-data (SIMD) architecture. Normally, GPU systems are viewed as SIMD machines to parallelize execution of thousands of threads which work on different sections of the data [**GLGN$^+$08**]. However, today's GPU systems are not entirely SIMD. For example, Nvidia Fermi architecture [**WKP11**] offers up to 512 cores, organized in 16 stream multiprocessor (SM) blocks each with 32 SIMD cores. Inside a block all cores must run the same code, but different blocks may run different codes. Cores located in different blocks may only communicate through the main memory. Therefore, one may view each SM block as one core capable of executing multiple threads of the same task but on different data [**HCW$^+$10**].

Throughout this manuscript, P denotes the total number of processor cores, p_k denotes the k'th processor ($1 \leq k \leq P$), and $f(k, k')$ denotes the FIFO channel from p_k to $p_{k'}$.

2.2. Dataflow Model

Let us briefly discuss the problems associated with thread-based models before explaining the dataflow model. When two threads try to access and modify a shared object, e.g., memory, the possible data race may result in corrupted values. Unfortunately, current synchronization solutions, e.g., locks, have a fundamental problem. They are not composable, meaning that combination of two correct lock-based codes does not necessarily result in a race-free code. In addition, locks may result in deadlock [**SL05**]. Multi-threading has been originally designed for server applications which naturally show process-level parallelism, while some desktop and embedded applications have fine-grained parallelism, complicated interactions, and pointer-based data structures which make them hard to parallelize with threads [**SL05**]. A number of leading experts believe that thread-based application development in general, is not a productive and reliable method of developing concurrent software [**Lee06**]. One reason is that concurrency constructs in Von Neumann languages introduce nondeterminism into programs [**Wen75**].

Dataflow models address the scalability and productivity issues of parallel threads [**Lee06**]. Here, the streaming application is described as a set of explicitly-parallel communicating tasks (actors). Tasks are atomic, i.e., their corresponding computation is specified in sequential semantics, and hence, intra-task parallelism is not exploited.

Synchronous dataflow (SDF) is a special type of dataflow in which tasks' data rates are specified statically [**LM87a**]. SDF-compliant kernels are at the heart of many streaming applications [**LM87a, TKA02, GB04**] and form the focus of our work. In the SDF model, a task is a tuple

$$(In, Out, H) \tag{2.1}$$

where $In \subseteq InputPorts$ is the set of input ports, $Out \subseteq Outputports$ is the set of output ports, and H denotes the transformation function of the task. $Ports = InputPorts \cup Outputports$, and $InputPorts \cap Outputports = \varnothing$. Each port has a statically-defined data rate, which is the mapping

$$rate : Ports \rightarrow \mathbb{N} \tag{2.2}$$

The application is modeled as a directed graph $G(V, E)$, known as task graph, where vertices $v \in V$ represent tasks, and directed edges $e \in E \subset Ports^2$ represent data communication channels. Each port is connected to exactly one channel, and each channel connects an output port of some task to an input port of another task.

A task can be executed (fired) upon availability of sufficient data on all its input ports. Firing of a task consumes data from its input ports, and produces data on its output ports. In streaming applications, the execution is meant to continue for infinite number of rounds.

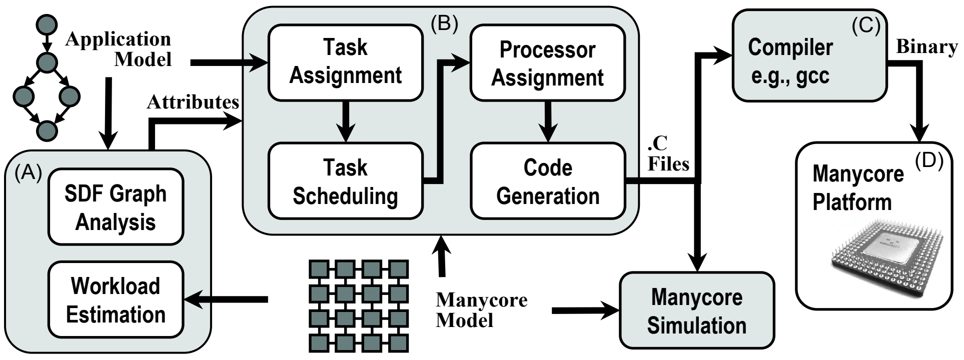


FIGURE 2.1. The big picture of software synthesis based on given application and target manycore models. A) SDF graph analysis and workload estimation steps provide the required information for B) automated synthesize of parallel software modules, which can be subsequently C) compiled, and then D) executed on the target manycore platform.

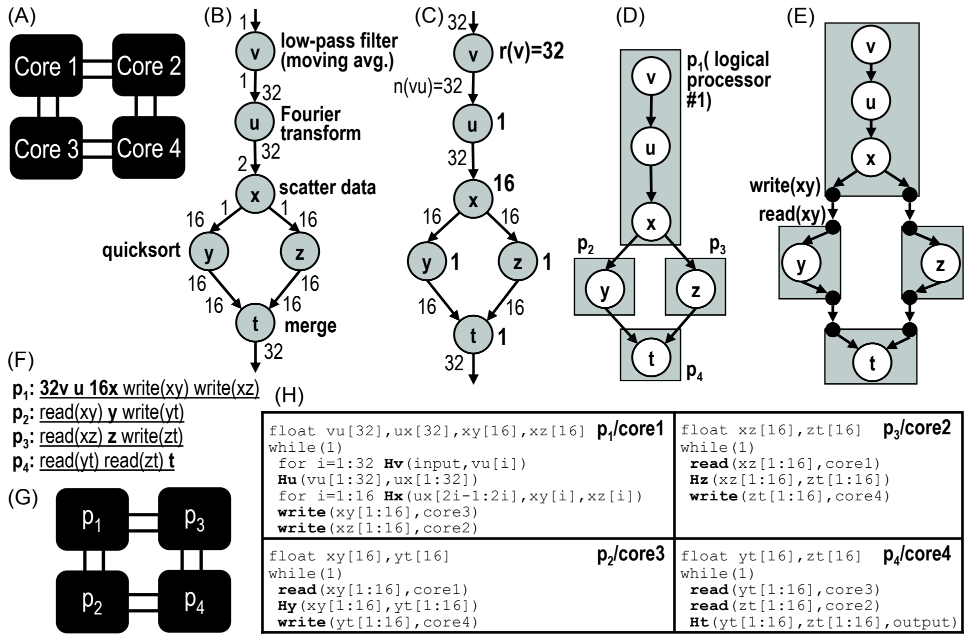
Figure 2.2.B on page 8 illustrates an example SDF model for a toy streaming application that calculates the Fourier transform of a filtered data and subsequently sort the result. The sort is performed by splitting the data in half, sorting each piece with quicksort algorithm and finally merging the two sorted pieces into the final sorted array using the well-known mergesort algorithm [**CLRS01b**].

2.3. Software Synthesis

Given the abstract model of a target manycore architecture and the SDF graph model of an streaming application, parallel software modules are automatically synthesized. As an intentional result of the architecture and application models, synthesized software modules need to directly send and receive messages to synchronize.

The main objective of software synthesis is to maximize throughput which is an important quality metric in the streaming application domain. Other optimization objectives and/or constraints include, but not limited to, judicious use of memory and communication resources on chip.

As shown in Figure 2.1.B, the software synthesis procedure involves several key steps including task assignment, task scheduling, processor assignment and code generation. In order to produce high quality code, such steps require a number of *attributes* for vertices (tasks) and edges (channels) of the SDF graph. As shown in Figure 2.1.A, SDF graph analysis and workload estimation steps provide this information. Once the parallel software



```
float vu[32],ux[32],xy[16],xz[16]
while(1)
 for i=1:32 Hv(input,vu[i])
 Hu(vu[1:32],ux[1:32])
 for i=1:16 Hx(ux[2i-1:2i],xy[i],xz[i])
 write(xy[1:16],core3)
 write(xz[1:16],core2)
```

```
float xz[16],zt[16]
while(1)
 read(xz[1:16],core1)
 Hz(xz[1:16],zt[1:16])
 write(zt[1:16],core4)
```

```
float xy[16],yt[16]
while(1)
 read(xy[1:16],core1)
 Hy(xy[1:16],yt[1:16])
 write(yt[1:16],core4)
```

```
float yt[16],zt[16]
while(1)
 read(yt[1:16],core3)
 read(zt[1:16],core2)
 Ht(yt[1:16],zt[1:16],output)
```

FIGURE 2.2. Example: A) Sample system with 4 processor cores. B) Sample SDF graph for a toy streaming application. C) SDF graph analysis. D) Task assignment. E) Inserting write and read tasks. F) Task scheduling. G) Processor assignment. H) Code generation.

modules are synthesized they can be executed on the target manycore platform (Figure 2.1.D), or alternatively simulated with a manycore simulator. To execute the application, each software module, e.g., `C` code, should be compiled to a binary code using the target native compiler, e.g., `gcc` (Figure 2.1.C).

The software synthesis steps in Figure 2.1 are briefly introduced here using a simple example. Figure 2.2.A illustrates an example target platform with 4 processor cores. Figure 2.2.B illustrates a toy streaming application that calculates the Fourier transform of a filtered data and subsequently sort the result. The sort is performed by splitting the data in half, sorting each piece with quicksort algorithm and finally merging the two sorted pieces into the final sorted array using the well-known mergesort algorithm as described in [**CLRS01b**]. The parallel software modules shown in Figure 2.2.H are produced by performing the following steps.

2.3.1. SDF Graph Analysis. As mentioned above, SDF graph analysis and workload estimation steps provide the graph attributes (Figure 2.1.A). Reader may skip the discussion on the SDF graph attributes and go directly to section 2.3.3 on page 11. The first attribute is *firing repetition* of tasks. It is defined as the number of times that a task $v \in V$ is fired in every round of the periodic application execution, and is denoted with $r(v)$. Formally

$$r : V \rightarrow \mathbb{N} \tag{2.3}$$

Channels can have different production and consumption rates. For example in Figure 2.2.B, channel vu has a production rate of 1 but a consumption rate of 32. As a result, the firing repetition $r(v)$ should be different for different tasks. In an SDF graph, $r(v)$ can be calculated solely based on the production and consumption rates as described in [**LM87b**] in order to satisfy the following condition

$$\forall e(v,u) \in E : r(v) \times rate(v.port) = r(u) \times rate(u.port) \tag{2.4}$$

where *v.port* and *u.port* denote the two ports that are connected via channel e. For example as shown in Figure 2.2.C, the rates are assigned such that $r(v) \times 1 = r(u) \times 32$. In this example, task v is fired 32 times, task x, 16 times, and all other tasks only once in each round of the periodic execution.

The second attribute is *communication volume* of channels $e(v,u) \in E$. It is defined as the total number of tokens transfered from the producer task v to the consumer task u in every round of the periodic application execution. The communication volume of channel e is denoted with $n(e)$ and can be determined based on the task firing repetitions as the following

$$\begin{gathered} n : E \rightarrow \mathbb{N} \\ \forall e(v,u) \in E : n(e) = r(v) \times rate(v.port) \end{gathered} \tag{2.5}$$

For example as shown in Figure 2.2.C, the communication volume of channel vu is equal to $n(vu) = 32$.

2.3.2. Workload Estimation. Another attribute is the *workload* of a task $v \in V$ on processor p_k, which is denoted by $w_k(v)$. Formally,

$$\begin{gathered} w_k : V \rightarrow \mathbb{N} \ (k \in [1,P]) \\ w_k(v) = r(v) \times latency \end{gathered} \tag{2.6}$$

where the latency denotes the execution latency of one firing of task v on processor p_k. The workload is inherently input-dependent, due to the strong dependency of the tasks' control flow (the transfer function H) with their input data. For example, the execution latency of the quicksort algorithm on a list would partially depend on the ordering of the numbers in the list. Other factors affecting the workload include the optimizations performed during compilation of synthesized software modules, and also, the existence of non-deterministic architecture units such as cache.

The accuracy of task workload estimates is crucial to the quality of software synthesis process specially the task assignment step. Therefore, the estimation method should be carefully designed based on a specific situation. Details of the workload estimation method used in this work are discussed in later chapters. In general, there are two main approaches to workload estimation, namely profiling and code analysis. Profiling-based approached measure the execution latency of tasks at runtime. Therefore, no analysis of the code is required, and also, the effects of compiler optimizations and non-deterministic hardware units such as cache and processor pipeline are better captured. Code analysis, however, can be easily automated to quickly analyze the workload of tasks at compile time without requiring to execute the tasks on the target platform [**Sar89, WEE$^+$08**]. Unpredictable effects of cache is often higher than the effect of processor pipeline or compiler optimizations. Scratchpad memories for embedded systems have been proposed to replace cache in order to improve predictability and energy consumption [**UDB06, NDB09**].

The last attribute is memory requirement of tasks and channels which is estimated as well. $m_k(v)$ refers to the amount of memory required by processor p_k for instructions of a task $v \in V$. For example in Figure 2.2, $m_k(y)$ is the memory footprint of the quicksort algorithm in processor p_k. Similarly, $m_k(e)$ refers to the amount of memory required to allocate the array which stores messages of a channel $e \in E$. For example as shown in

Attribute	Notation	Possible Unit
firing repetition	$r(v)$	-
number of tokens	$n(e)$	-
computation workload	$w_k(v)$	msec.
required memory	$m_k(v), m_k(e)$	KB

FIGURE 2.3. The attributes of vertices (tasks) and edges (channels) of an SDF graph $G(V, E)$, which is provided by the steps shown in Figure 2.1.A

Figure 2.2.H, $m_1(vu)$ is equal to the amount of memory allocated for tokens of channel vu, i.e., $m_1(vu) = n(vu) \times \text{sizeof}(float) = 32 \times 4$. Estimating the memory requirement is often simple, specially for the embedded streaming applications because they normally do not dynamically allocate data structures of unpredictable size. Figure 2.3 summarizes the above attributes for future reference.

2.3.3. Task Assignment. Every task should be assigned to a processor for execution. At this point, the physical location of processors on the chip is ignored [**SK03**]. In other words, tasks are assigned to logical processors in this step, and later in the processor assignment step, logical processors are assigned to the physical processors on the chip. Figure 2.2.D illustrates an example task assignment in which tasks u, v and x are assigned to logical processor p_1, y to p_2, z to p_3 and t to p_4.

When the producer and consumer tasks of a channel are assigned to different processors, the tokens on this channel should be transfered using the platform FIFO channels. This is modeled by adding new write and read tasks to the graph. For example as shown in Figure 2.2.E, for channel xy from p_1 to p_2, new write and read tasks are added to the graph. Adding the new communication tasks greatly impacts the degree of freedom for task scheduling (more details on this subject in Section 2.3.4 on the next page).

The main objective of task assignment is to judiciously distribute the workload among the processors in order to maximize throughput. This requires the workload attributes $w_k(v)$, i.e., the workload of task v if it is allocated on processor p_k. Task assignment is simpler for manycore systems with identical cores [**G$^+$02**], while in heterogeneous manycore systems the difference among the cores should also be considered for efficient implementation [**VR99, SK03, MK08**]. Depending on the target architecture, task assignment may involve additional objectives and/or constraints. For example, minimization of interprocessor communication overhead is often a second optimization objective specially since it normally affects the throughput [**SBGC07, HG09**]. This requires the use of attributes $n(e)$. Another example is when onchip memory is limited, in which case, the memory requirement of each processor can be added as a constraint. This requires the use of attributes $m(v)$ and $m(e)$. If the required memory exceeds the available local memory the implementation is infeasible, and hence, the estimated performance gain through workload balancing is irrelevant. However, most task assignment approaches focus mainly on workload in order to increase the throughput and consider the memory limits only as a second constraint if at all. This is because there are often a number of ways to spill data in and out of the limited local memory [**UDB06, HCK$^+$09**], but note that such approaches often work only for the data memory and not the instruction memory.

Task assignment for a manycore target with P processors can be formulated as a graph partitioning problem in which the graph should be partitioned into P subgraphs. An example partitioning objective is to evenly distribute weights of the vertices (task workloads) while minimizing weight of cut edges (communications) [**MK08, HG10**]. Other approaches include ILP-based task assignment formulations which normally provide higher degree of freedom in defining a heterogeneous architecture but may take longer to execute [**HCK$^+$09**]. Heuristic task assignment methods often sort the tasks first and allocate them on the processors in that order. The criteria for sort is normally either the task workloads alone or a combincation of workloads, communication overheads and memory requirement [**G$^+$02, SBGC07**].

2.3.4. Task Scheduling. Assuming each core executes only one thread, which is the case in our hardware model, tasks that are assigned to the same processor should be ordered for sequential execution on that processor. In SDF model, the production and consumption rates are known at compile-time and hence such ordering can be determined statically [**LM87b**]. Figure 2.2.F shows a valid schedule for each of the logical processors in our example. Note that the execution is periodic, for example the schedule [read(xy), y, write(yt)] for processor p_2 means the following sequence repeats periodically. The data on channel xy is first read from FIFO, task y is executed, and then the data produced on channel yt are written to FIFO.

Communications between tasks that are assigned to different processors cause interprocessor dependencies, and hence, scheduling of the tasks on one processor is not independent from scheduling of the tasks on other processors. Failure to consider this effect may result in a deadlock situation [**ZL06**]. Task scheduling should avoid deadlock by considering the dependency of a task in every processor with the tasks in other processors. This is closely related to the effect of FIFO channels especially when cycles exist in the dataflow graph [**GGB+06, SGB08, LGX+09, PD10**].

Besides deadlock avoidance, the main optimization objective of task scheduling, however, is to maximize throughput by judiciously scheduling the tasks in time. Inter-processor dependencies should be carefully analyzed at compile-time in order to avoid idle periods in the execution. Iteration overlapping, a.k.a. pipeline parallelism [**GTA06, KM08**], is a technique that alters the task schedule by overlapping the execution of different iterations of the streaming application in order to fill the idle gaps. This is closely related to software pipelining where multiple iterations of a loop are overlapped [**Rau94**]. Insertion of new write and read communication tasks to the task graph (Section 2.3.3) provides more room for such optimizations because it disentangles the execution of a task from its dependent consumers (or producers) in other processors, and it also disentangles the consumers of the task from one another.

As opposed to task assignment where throughput and memory requirements are often considered as separate factors, in task scheduling the two have a strong relation [**BLM96, BML99**]. In other words, the schedule of tasks directly impacts both the throughput and the memory requirement at the same time. The reason for the impact on memory is that the local buffering requirement of a channel vary depending on how far its producer or consumer tasks are located in the schedule of that processor, and how many times the task executes on every iteration of the periodic execution [**ZTB00, KTA03, KMB07**].

Hence, memory requirement may be taken into account as a constraint when onchip memory is limited. Note that when tasks are coarse-grain, e.g., functions, as opposed to fine-grain, e.g., instructions, multiple calls to the function are inserted into the code instead of multiple copies of the same instruction sequence. In other words, the impact of scheduling on instruction memory is limited for coarse-grain task graphs. Data memory requirement, however, has a strong dependency to the schedule as mentioned above. Therefore, when onchip memory is limited, it would be a wise option to consider instruction memory requirements during task assignment, and data memory requirements during task scheduling.

2.3.5. Processor Assignment. In the task assignment step, tasks are assigned to logical processors, and in this step which is also known as layout or processor binding, the logical processors are assigned to physical processors on the chip [**G$^+$02, HM05**]. This approach simplifies the problem of assigning tasks to the processors by dividing it in two sub-problems. Hence, faster and more efficient optimization methods could be designed for each of the two steps. Processor assignment considers the effect of onchip communications on the throughput and try to assign communicating processors as close to each other as possible to avoid network congestion. The optimizations required in this step highly depends on the underlying architecture of the onchip network. For example, in manycore systems like AsAP [**TCM$^+$08**] where onchip communication resources are limited sophisticated algorithms are required to achieve a feasible and high throughput implementation.

In certain cases, task assignment may also need to be aware of the limitations in the onchip network. For example in the first version of AsAP [**YMA$^+$06**], each processor could communicate with only its four neighbor processors. Task assignment for such platforms should consider this limitation in grouping the tasks, or otherwise it may not be possible for the processor assignment step to find a feasible solution.

An alternative way, however, would be to combine task assignment and processor assignment in one step. This is more favorable when the target manycore system has a high degree of heterogeneity. Task assignment needs to be aware of the exact processor architecture in order to have a workload estimate of that task or its memory size constraints. If the degree of heterogeneity is low it may still be favorable to have separate task assignment and processor assignment steps.

2.3.6. Code Generation. Task functionalities are provided as sequential computations that are kept intact throughout the synthesis process. The software code for each processor is synthesized by stitching together the set of tasks that are assigned to that processor according to their schedule. For tasks that are assigned to the same processor, inter-task communication is implemented using arrays. That is, the producer task writes its data to an array, which is then read by the consumer task. Inter-processor communication is implemented using the new write and read tasks inserted in the task assignment step. Figure 2.2.H on page 8 illustrates an example.

Decades of effort have been put into design of efficient sequential compilers such as `gcc` [**BGS94**]. To harness the power of optimization algorithms in sequential uni-processor compilers, the code generation only produces a source code, e.g., `C` file, for each processor core. A native compiler of the target platform can later compile this code into binary. During the uni-processor compilation, specific I/O instructions or calls to certain OS services or library functions may need to be inserted into the binary code to implement the inter-processor communications.

Task workloads, task schedules, production and consumption rates and many other attributes of the automatically synthesized software modules are known at this point. This enables efficient deployment of source-level optimizations which otherwise would not be possible or very difficult and inefficient. Buffer merging is a technique which reduces the data memory requirement. Through careful analysis of the time intervals at which buffers are alive, the source code is transformed such that allocated memory of the buffers overlap [**MB04, FHHG10, FHHG11**]. In order to reduce energy consumption, workload and scheduling information may assist automatic insertion of DVS (dynamic voltage scaling) instructions into the code [**LBDM02, IHK04, FRRJ07**].

CHAPTER 3

Task Assignment Techniques for Pipelined Execution of Stream Programs

The main objective of task assignment is to optimally distribute the workload among the processors in order to maximize the application throughput. Depending on the target architecture, task assignment may involve additional objectives and/or constraints including but not limited to judicious use of limited onchip memory or minimization of inter-processor communication.

In this chapter, we first present a task assignment method for heterogeneous soft dual-processor systems. The objective is to maximize throughput, under the constraint that generated code for a processor meets processor's instruction and data memory size constraints, if needed. Given the high-degree of heterogeneity in the system, differerent architectures are likely to impact application throughput differently. Optimal mapping of application tasks onto processors depends on the specifics of the architectures. Ideally, one would want to have a versatile software synthesis solution that can be parameterized to target different configurations. Specifically, we present a formally-versatile dual-processor task assignment algorithm whose formal properties hold for, and hence it is applicable to, a variety of target hardware platforms with different implementation-driven objectives and constraints. Leveraging the graph theoretical properties of applications task graphs, we develop a provably-effective task assignment algorithm that both maximizes throughput, and guarantees meeting instruction and data size constraints according to high-level estimates. Measurement of generated code size and throughput of emulated systems validate the effectiveness of our approach [**HHG07, HG10**].

Next, based on the above dual-processor algorithm, we develop a heuristic method for pipelined execution on targets with an arbitrary number of identical processors. In other words, we view the target platform as a chain of processors, and iteratively apply the graph bi-partitioning method (dual-processor task assignment) to partition the task graph among all the processors. Throughput measurement of the generated code on emulated systems validate the effectiveness of our approach [**HG08, HG09**].

3.1. Terms and Definitions

Task assignment for dual-processor platforms can be viewed as bi-partitioning of application task graph, in which tasks in the same partition are assigned to the same processor. A partitioning of graph $G(V, E)$ is constructed by removal (cutting) of a subset of edges (channels) to create two connected subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. We use the term cut C to refer to such cut channels. Formally

$$\begin{aligned} & V_1 \cup V_2 = V \\ & E_1 \cup E_2 \cup C = E \end{aligned} \tag{3.1}$$

As discussed in the previous chapter (Figure 2.3 on page 10), the task graph is annotated with a number of attributes. Figure 3.1 shows an example task graph in which all the attributes w_1, w_2, n, m_1 and m_2 are equal to 1, except for those shown in the figure. Later in Section 3.5, we discuss extensions to handle other attributes, if necessary. The attributes are naturally extended to partitions in the following manner.

Computation: The total workload of tasks, excluding the extra write and read tasks, assigned to processor p_k, i.e., in a partition G_k, is equal to[1]

$$\|w_k(V_k)\| = \sum_{v \in V_k} w_k(v) \tag{3.2}$$

For example in Figure 3.1, $V_1 = \{a, b, c\}$ and thus $\|w_1(V_1)\| = w_1(a) + w_1(b) + w_1(c) = 1 + 1 + 2 = 4$.

Communication: If a channel $e(u, v)$ is cut, tasks u and v are assigned to different processors, and thus the corresponding communication between u and v has to pass through the on-chip network. The term $\|n(C)\|$ refers to the total number of messages transfered over all the cut channels. In Figure 3.1, $C = \{bd, ce\}$ and thus $\|n(C)\| = n(bd)+n(ce) = 1+1 = 2$.

[1]Throughout this manuscript, for a function $f : X \to \mathbb{N}_0$, the term $f(X)$ denotes the set of $f(x)$ values for every x, i.e., $f(X) = \{f(x)|x \in X\}$. For example, $w(V_k) = \{w(v)|v \in V_k\}$. The term $\|f(X)\|$ denotes the L_1 norm of $f(X)$ which is equal to $\sum_{x \in X} f(x)$. For example, $\|w(V_k)\| = \sum_{v \in V_k} w(v)$. The term $|X|$ denotes the number of elements in a set X.

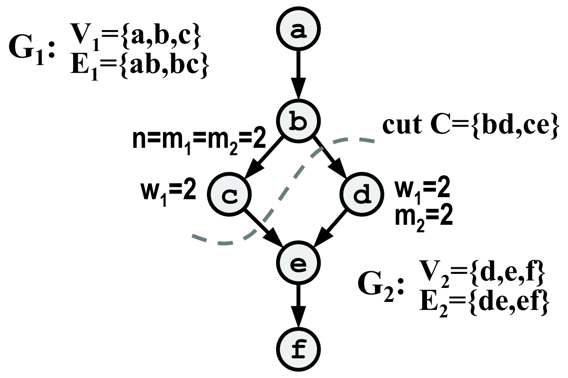


FIGURE 3.1. An example task graph in which all the attributes w_1, w_2, n, m_1 and m_2 are equal to 1, except for those shown in the figure. Cut $C = \{bd, ce\}$ divides the graph G in two partitions G_1 and G_2.

$\|w_k(V_k)\| = \sum_{v \in V_k} w_k(v)$	computation workload assigned to processor p_k
$\|n(C)\| = \sum_{e \in C} n(e)$	# of message transferred between processor 1 and 2
$\|m_k(V_k)\| = \sum_{v \in V_k} m_k(v)$ $\|m_k(E_k)\| = \sum_{e \in E_k} m_k(e)$ $\|m_k(C)\| = \sum_{e \in C} m_k(c)$	memory required by processor p_k for: - tasks - inter-task intra-processor communications - inter-processor communications

FIGURE 3.2. Partition attributes are natural extension of vertex and edge attributes.

Memory: Cut channels demand allocation of data memory on both processors. The term $\|m_k(C)\|$ refers to the amount of data memory allocated for the cut channels on processor p_k. In Figure 3.1, $\|m_1(C)\| = m_1(bd) + m_1(ce) = n(bd) \times \text{sizeof}(datatype) + n(ce) \times \text{sizeof}(datatype) = 1 + 1 = 2$.

The rest of the channels allocate memory in only one processor. The term $\|m_k(E_k)\|$ refers to the total amount of memory required by processor p_k for implementing intra-processor communication, in which E_k represents the subset of channels that are allocated to processor p_k. In Figure 3.1, $E_1 = \{ab, bc\}$, and thus, $\|m_1(E_1)\| = m_1(ab) + m_1(bc) = 1+2 = 3$. Similarly, the term $\|m_k(V_k)\|$ denotes the amount of instruction memory required by processor p_k for task set V_k. The partition attributes are summarized in Figure 3.2 for future reference.

3.1.1. Problem Statement. We target task graphs that contain no cycles. If a given task graph contains a cycle, we collapse the vertices in the cycle into a single task to represent the application as a directed acyclic graph (DAG). Such task graphs form an important subset, because many important streaming kernels can be represented in this fashion. Hence, we aim to assign tasks to processors such that one processor would always send and the other would always receive message. This uni-directional flow of data intuitively suits pipelined throughput-driven execution of streaming applications In the graph domain, this translates to *convex* cuts, which refer to cuts that connect vertices in G_1 to vertices in G_2 (and not the other way around). Convex cuts, which form the focus of our work in this chapter, characterize a subset of possible task assignments. If all task assignment possibilities were to be considered, task scheduling had to be combined with task assignment to allow evaluation of candidate solutions.

Tasks in the partition G_k will be assigned to, and executed by processor p_k, and messages on cut channels $e \in C$ will be transfered between processors using the onchip network. Hence, task assignment for dual-processors can be formally defined as the following optimization problem:

$$\begin{aligned}
&a)\text{ minimize:}\\
&Q = F\big(\|w_1(V_1)\|, \|w_2(V_2)\|, \|n(C)\|\big)\\
&b)\text{ constraint:}\\
&\|m_1(V_1)\| + \|m_1(E_1)\| + \|m_1(C)\| \leq \text{memory of processor 1}\\
&\|m_2(V_2)\| + \|m_2(E_2)\| + \|m_2(C)\| \leq \text{memory of processor 2}
\end{aligned} \tag{3.3}$$

That is, the goal is to find the cut C which, (a) maximizes throughput and, (b) meets the processors' memory size constraints. The cost function Q models application execution period, which is the inverse of its throughput. The execution period depends on processor workloads and inter-processor communication. The specific relation, however, depends on the underlying hardware. Our goal is to devise a versatile method to handle a wide variety of cost functions.

For example in a system with negligible interprocessor communications, it may be accurate enough to estimate the execution period as $Q = F(\|w_1(V_1)\|, \|w_2(V_2)\|) = \max\{\|w_1(V_1)\|$ $, \|w_2(V_2)\|\}$, because the pipeline throughput would be limited by the slowest of the two processors. This simplified cost function promotes balancing processors' workload, which has been the focus of most classical task assignment schemes. As another example, assume a sufficiently-buffered FIFO link between the two processors implements the virtual network. A reasonable estimation function would be $Q = \max\{\|w_1(V_1)\| + \alpha_1\|n(c)\|, \alpha_2\|n(c)\| + \|w_2(V_2)\|\}$, where α_k is the extra cycles in workload of processor p_k to push (pop) one unit of data to (from) the FIFO.

In practice, out of two solutions with identical workload distributions, the one with smaller inter-processor communication is always preferred. Therefore, without loss of generality we assume that the cost function Q is non-descending in $\|n(c)\|$.

3.2. Attribute Properties and Transformations

In this section, we exploit structural properties of task graphs to develop a transformation on the attributes, that assists us in quick evaluation of a cut. Let us assume an imaginary attribute θ_1 is assigned to both tasks (vertices) and channels (edges) in the graph $G(V, E)$. Our objective is to transform θ_1 to a new set of attributes, θ'_1, assigned only to edges of G, such that their summation on any cut would give the summation of the original θ_1 attributes on vertices and edges in partition 1.

Let e_i and e_o denote the set of incoming and outgoing edges of a vertex, respectively. Let e^*_o be a randomly-selected outgoing edge. The transformed attributes θ'_1 for outgoing edges of a vertex are recursively defined as follows:

$$\forall v \in V \ : \ \theta'_1(e_o) = \begin{cases} \sum_{e_i} \theta'_1(e_i) \ + \theta_1(v) + \theta_1(e_o) & \text{if } e^*_o \\ \theta_1(e_o) & \text{otherwise} \end{cases} \tag{3.4}$$

Figure 3.3 shows an example. Intuitively speaking, attribute transformation works similar to gravity. The task graph can be viewed as a physical structure in which every element has a weight, i.e., $\theta_1(v)$ and $\theta_1(e)$ can be viewed as the weight of vertex v and edge e, respectively. Hence, the random selection of one outgoing edge is analogous to disconnecting the corresponding joint in the structure. That is, every vertex stays connected to exactly one of its outgoing edges in the physical domain. The amount of weight held by edge e determines the value of $\theta'_1(e)$. In Figure 3.3, edge ab holds $\theta'_1(ab) = \theta_1(a) + \theta_1(ab)$ of weight. Vertex b in the figure is disconnected from edge bc, and therefore, total weight of vertices a

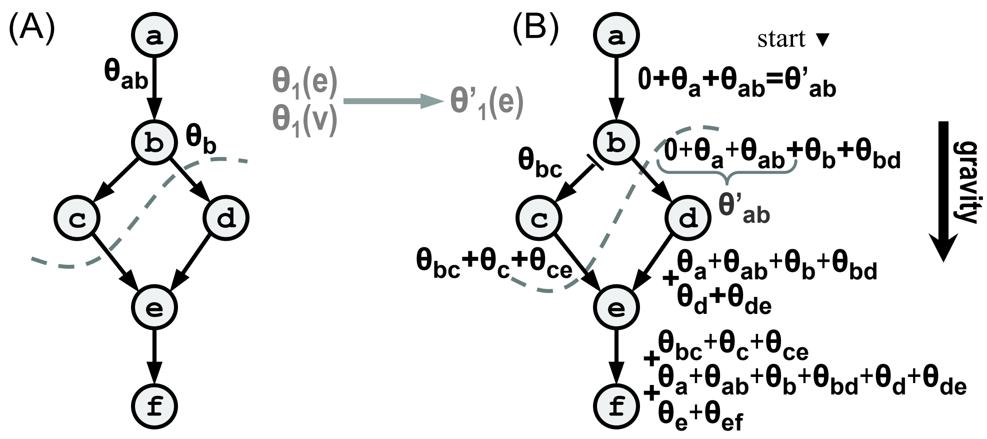


FIGURE 3.3. A) Sample task graph with attributes θ_1 for vertices and edges. B) The task graph after transforming θ_1 to θ_1'.

and b and edges ab and bd is held by edge bd. Note that this joint disconnection is only an intuitive analogy, i.e., none of the edges are actually removed or disconnected.

As expected from the gravity analogy, the transformation satisfies our objective property. That is, the sum of θ_1' attributes on cut C is equal to the sum of θ_1 attributes on the edges and vertices of the top partition of graph, $G_1(V_1, E_1)$. Intuitively, the cut edges have to hold the entire weight above them. In Figure 3.3, $\theta_1'(bd) + \theta_1'(ce)$ is equal to $\{\theta_1(a) + \theta_1(b) + \theta_1(c)\} + \{\theta_1(ab) + \theta_1(bc)\} + \{\theta_1(bd) + \theta_1(ce)\}$.

LEMMA 3.1. *The transformation propagates the attribute of an arbitrary vertex (edge) along exactly one directed path, referred to as the* propagation path, *from the vertex (edge) to the unique sink vertex.*

PROOF. Let a be an arbitrary vertex in the directed acyclic graph. The transformation propagates $\theta_1(a)$ along exactly one of its outgoing edges, which is selected at random. Let b be the destination vertex of the randomly-selected outgoing edge. If b is the sink vertex, then the lemma is proved. Otherwise, $\theta_1(a)$ is propagated along exactly one of the outgoing edges of b, and the same argument can be made iteratively. Since graph is acyclic, we will never visit a vertex that we have visited before. The graph has finite number of vertices and hence, the iteration will have to end by arriving at the sink vertex. Similar argument can be made for edge attributes. □

THEOREM 3.2. *For a convex cut on a directed acyclic graph*

$$\underbrace{\sum_{v\in V_1}\theta_1(v)}_{\substack{\textit{weight of}\\ G_1\textit{ vertices}}} + \underbrace{\sum_{e\in E_1}\theta_1(e)}_{\substack{\textit{weight of}\\ G_1\textit{ edges}}} + \underbrace{\sum_{e\in C}\theta_1(e)}_{\substack{\textit{weight of}\\ \textit{cut edges}}} = \underbrace{\sum_{e\in C}\theta_1'(e)}_{\substack{\textit{total weight}\\ \textit{held by cut } C}} \tag{3.5}$$

The theorem should be intuitive from the gravity analogy, in which the weight of every vertex or edge in G_1 is held by one and only one of the cut edges. For example in Figure 3.3, weight of vertex a (i.e., $\theta_1(a)$) is held by the cut edge bd, and not by ce. Therefore, across all the cut edges, weight of vertices and edges in G_1 are considered exactly once.

PROOF. We argue that C and the propagation path of an arbitrary vertex $a \in G_1$ intersect at exactly one edge. If C and the propagation path of vertex a do not intersect, then $a \in G_1$ is connected to the sink vertex via a connected path and hence, C is not a cut. If they intersect in more than one edge, then the propagation path direction goes out of G_1 and then back into G_1, which means that C is not convex. Therefore, the two edge set intersect at exactly one edge, and hence, $\theta_1(a)$ is accurately captured in θ_1' of the edge. Similar arguments can be made for arbitrary edges in G_1 or C. □

We also introduce another transformation similar to above, with the minor difference that the gravity is replaced with a force away from (as opposed to toward) the bottom of the graph. In other words, the graph can be temporarily held upside down. Here, we convert θ_2 attributes to a new set of θ_2' attributes assigned to the edges. Figure 3.4 shows an example.

THEOREM 3.3. *Similarly we have:*

$$\underbrace{\sum_{v\in V_2}\theta_2(v)}_{\substack{\textit{weight of}\\ G_2\textit{ vertices}}} + \underbrace{\sum_{e\in E_2}\theta_2(e)}_{\substack{\textit{weight of}\\ G_2\textit{ edges}}} + \underbrace{\sum_{e\in C}\theta_2(e)}_{\substack{\textit{weight of}\\ \textit{cut edges}}} = \underbrace{\sum_{e\in C}\theta_2'(e)}_{\substack{\textit{total weight}\\ \textit{held by cut } C}} \tag{3.6}$$

PROOF. Construct a new graph G^r from G by reversing the direction of all edges, and then, apply Theorem 1 to G^r. □

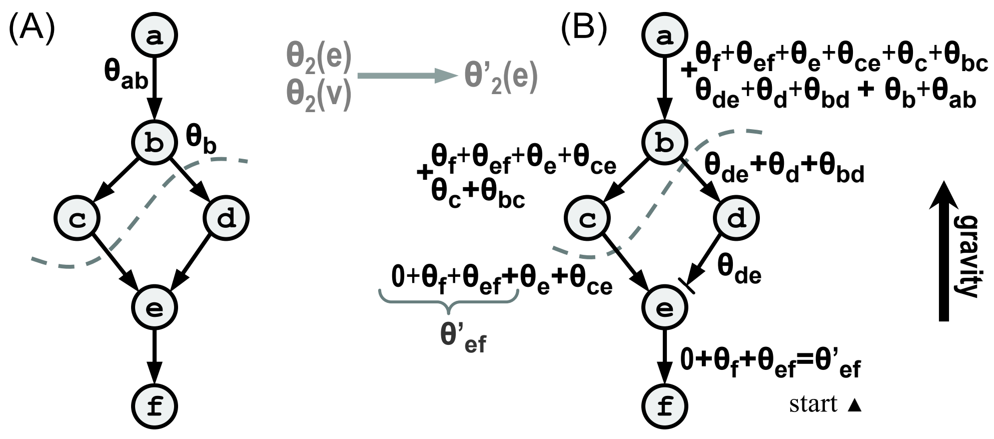


FIGURE 3.4. A) Sample task graph with attributes θ_2 for vertices and edges. B) The task graph after transforming θ_2 to θ_2'.

COROLLARY 3.4. *By combining Theorems 3.2 and 3.3 we have*

$$\sum_{v\in V_k}\theta_k(v)+\sum_{e\in E_k}\theta_k(e)+\sum_{e\in C}\theta_k(e)=\sum_{e\in C}\theta_k'(e) \tag{3.7}$$

which can be written as

$$\|\theta_k(V_k)\|+\|\theta_k(E_k)\|+\|\theta_k(C)\|=\|\theta_k'(C)\| \tag{3.8}$$

for $k\in\{1,2\}$.

3.3. Versatile Task Assignment via Graph Partitioning

We approach task assignment as a graph partitioning instance, and develop an algorithm that is provably optimal in minimizing cost function Q, or equivalently, maximizing pipeline throughput. The technique is versatile and can optimize a variety of cost functions inspired by different hardware configurations. For clarity, we present our technique using the attributes discussed in Section 3.1. Later in Section 3.5, we will present extensions to our approach.

3.3.1. Applying Attribute Transformation. As summarized in Figure 3.2, to obtain the value of partition attributes all vertices or edges in the partition have to be enumerated. For example, one has to enumerate all vertices in G_1 to calculate $\|w_1(V_1)\|$. It would be more efficient to evaluate Equation 3.3 by only processing attributes of the edges in cut C. Here we apply the attribute transformation, and as a result, we will have a new set of attributes (e.g., w'_1 instead of w_1), which have the desired property. The transformation is applied to the following attributes. Figure 3.5 provides a summary.

Computation: w_k attributes are transformed to a new set of attributes called w'_k. In this case, "θ" is mapped to "w", which means, $\theta_k(v) = w_k(v)$, $\theta_k(e) = 0$ and $\theta'_k(e) = w'_k(e)$. Based on Corollary 3.4, we have:

$$\|w_k(V_k)\| = \|w'_k(C)\| \tag{3.9}$$

For example in Figure 3.6.A, we need to sum up the workload of vertices a, b and c to calculate $\|w_1(V_1)\| = w_1(a)+w_1(b)+w_1(c)$ from the original graph. After the transformation (Figure 3.6.B), we can obtain $\|w_1(V_1)\|$ by enumerating edges bd and ce, i.e., $\|w'_1(C)\| = w'_1(bd) + w'_1(ce) = \{w_1(a) + w_1(b)\} + \{w_1(c)\}$. Thus, we use $\|w'_1(C)\|$ instead of $\|w_1(V_1)\|$.

Communication: Since $\|n(C)\| = \sum_{e\in C} n(e)$, it is already calculated by looking only at the cut edges, and thus, we do not need to apply the transformation to attributes n.

Memory: Attributes m_k are transformed to a new set of attributes called m'_k. Here, $\theta_k(v) = m_k(v)$, $\theta_k(e) = m_k(e)$ and $\theta'_k(e) = m'_k(e)$. Based on Corollary 3.4, we have

$$\|m_k(V_k)\| + \|m_k(E_k)\| + \|m_k(C)\| = \|m'_k(C)\| \tag{3.10}$$

For example, the memory requirement of processor 2 is originally calculated by visiting vertices f, e and d, and edges ef, de, bd and ce, i.e., $\|m_2(V_2)\| + \|m_2(E_2)\| + \|m_2(C)\| = \{m_2(f) + m_2(e) + m_2(d)\} + \{m_2(ef) + m_2(de)\} + \{m_2(bd) + m_2(ce)\}$ (Figure 3.6.A). After the transformation, we can derive the same value from $m'_2(bd) + m'_2(ce)$ (Figure 3.6.B). Thus we use $\|m'_k(C)\|$ instead of $\|m_k(V_k)\| + \|m_k(E_k)\| + \|m_k(C)\|$.

Following transforming the attributes, we have a new set of edge attributes, $n(e)$, $w'_1(e)$, $w'_2(e)$, $m'_1(e)$ and $m'_2(e)$, that are summarized in Figure 3.5. They enable evaluation of both the cost function and memory constraint by enumeration of the edges in cut C. Therefore, our target task assignment problem can be cast as finding the cut in the graph, subject to the following objective and constraints:

$$\begin{aligned} &a)\text{ minimize:} && Q(C) = F\big(\|w'_1(C)\|, \|w'_2(C)\|, \|n(C)\|\big) \\ &b)\text{ constraints:} && \|m'_1(C)\| \le \text{memory of processor 1} \\ & && \|m'_2(C)\| \le \text{memory of processor 2} \end{aligned} \tag{3.11}$$

$\theta_k \to \theta'_k$	Corollary 3.4	Calculation
$\theta_k(v) = w_k(v)$, $\theta_k(e) = 0$ $\quad \theta'_k(e) = w'_k(e)$	$\lVert w_k(V_k)\rVert = \lVert w'_k(C)\rVert$	$\sum_{e\in C} w'_k(e)$
transformation not required	$\lVert n(C)\rVert$	$\sum_{e\in C} n(e)$
$\theta_k(v) = m_k(v)$, $\theta_k(e) = m_k(e)$ $\quad \theta'_k(e) = m'_k(e)$	$\lVert m_k(V_k)\rVert + \lVert m_k(E_k)\rVert$ $+\lVert m_k(C)\rVert = \lVert m'_k(C)\rVert$	$\sum_{e\in C} m'_k(e)$

FIGURE 3.5. Applying attribute transformation to the application task graph.

To simplify the notation, we represent the attributes in an *attribute vector* $\vec{\beta}(e) = [w'_1(e), w'_2(e), m'_1(e), m'_2(e), n(e)]$. For example, $\beta_3(e) = m'_1(e)$. Hence, $\lVert\beta_3(C)\rVert = \lVert m'_1(C)\rVert = \sum_{e\in C} m'_1(e)$ (Figure 3.6.C). Therefore, Equation 3.11 can be formulated as:

$$\begin{aligned} &a)\text{ minimize:}\quad Q(C) = F\big(\lVert\beta_1(C)\rVert, \lVert\beta_2(C)\rVert, \lVert\beta_5(C)\rVert\big) \\ &b)\text{ constraints: } \lVert\beta_3(C)\rVert \le \beta_3^{\max} \text{ and } \lVert\beta_4(C)\rVert \le \beta_4^{\max} \end{aligned} \tag{3.12}$$

The term $\beta_3^{\max}$ (and $\beta_4^{\max}$) denotes the same value we had in Equation 3.11, i.e., available memory of processor 1 (and 2).

3.3.2. Graph Expansion. The number of convex cuts C in G can grow exponentially with respect to graph complexity. In order to tractably find the optimal path, we would need to eliminate paths that are guaranteed to yield inferior solutions from consideration.

For this purpose, we first planarize the task graph G using the transformation developed in [**HG09**]. Note that although some programming languages, such as StreamIt [**TKA02**], guarantee task graph planarity, our proposed method does not require a specific language.

Given a planar embedding of $G(V,E)$, we construct the well-defined dual graph $G^*(V^*,E^*)$. The β attributes of edges $e \in E$ are transferred to the corresponding edges $e^* \in E^*$ (Figure 3.6.D). A simple path P^* from vertex s^* to t^* in graph G^* identifies a convex cut C in graph G. Therefore, the dual graph enables us to evaluate the quality of a task assignment by evaluating the quality of the corresponding path from s^* to t^* in G^*. That is, since $\lVert\beta_i(C)\rVert = \sum_{e\in C}\beta_i(e) = \sum_{e^*\in P^*}\beta_i(e^*) = \lVert\beta_i(P^*)\rVert$, we evaluate Equation 3.12 from P^* instead

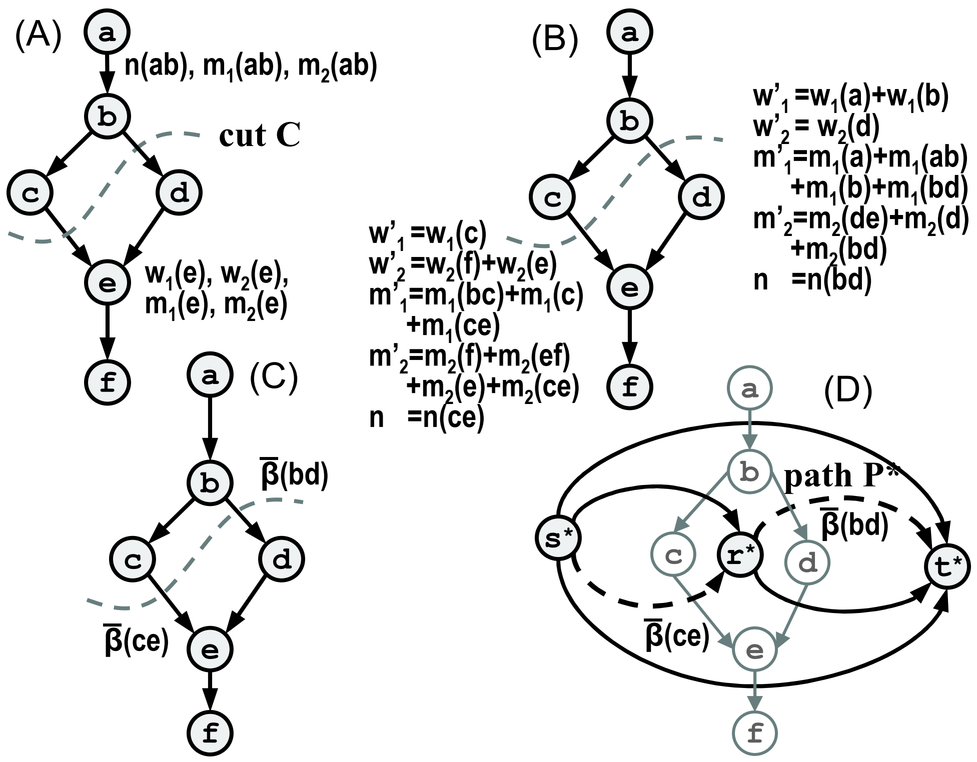


FIGURE 3.6. A) Task graph $G(a, f)$ and a sample cut C. B) New attributes after the transformation. C) Vector representation of the attributes. D) Dual graph $G^*(s^*, t^*)$, and path P^*.

of C:

$$\begin{aligned} &a)\text{ minimize:} \quad Q(P^*) = F\big(\|\beta_1(P^*)\|, \|\beta_2(P^*)\|, \|\beta_5(P^*)\|\big) \\ &b)\text{ constraints: } \|\beta_3(P^*)\| \le \beta_3^{\max} \text{ and } \|\beta_4(P^*)\| \le \beta_4^{\max} \end{aligned} \tag{3.13}$$

Next, we expand the dual graph G^* to a graph $G^\dagger$, constructed in four dimensional (4-D) space, to better visualize the situation (Figure 3.7). Each dimension of $G^\dagger$ represents one element of the attribute vector, e.g., workload of the top partition. For every vertex v^* in G^*, we have vertices $v^*[i_1, i_2, i_3, i_4]$ in graph $G^\dagger$ in 4-D space. For example in the figure, vertex r^* is expanded into two vertices $r^*[0, 3, 2, 7]$ and $r^*[2, 2, 4, 4]$ (4-D indices of the two vertices are in the middle column of Figure 3.7.D). Graph $G^\dagger$ is constructed such that the information stored in $\vec{\beta}(e^*)$ attributes in graph G^*, is represented in the structure of graph $G^\dagger$. Formally, vertex $u^*[\vec{i}]$ (abbreviation for $u^*[i_1, i_2, i_3, i_4]$) is connected to vertex $v^*[\vec{j}]$ in

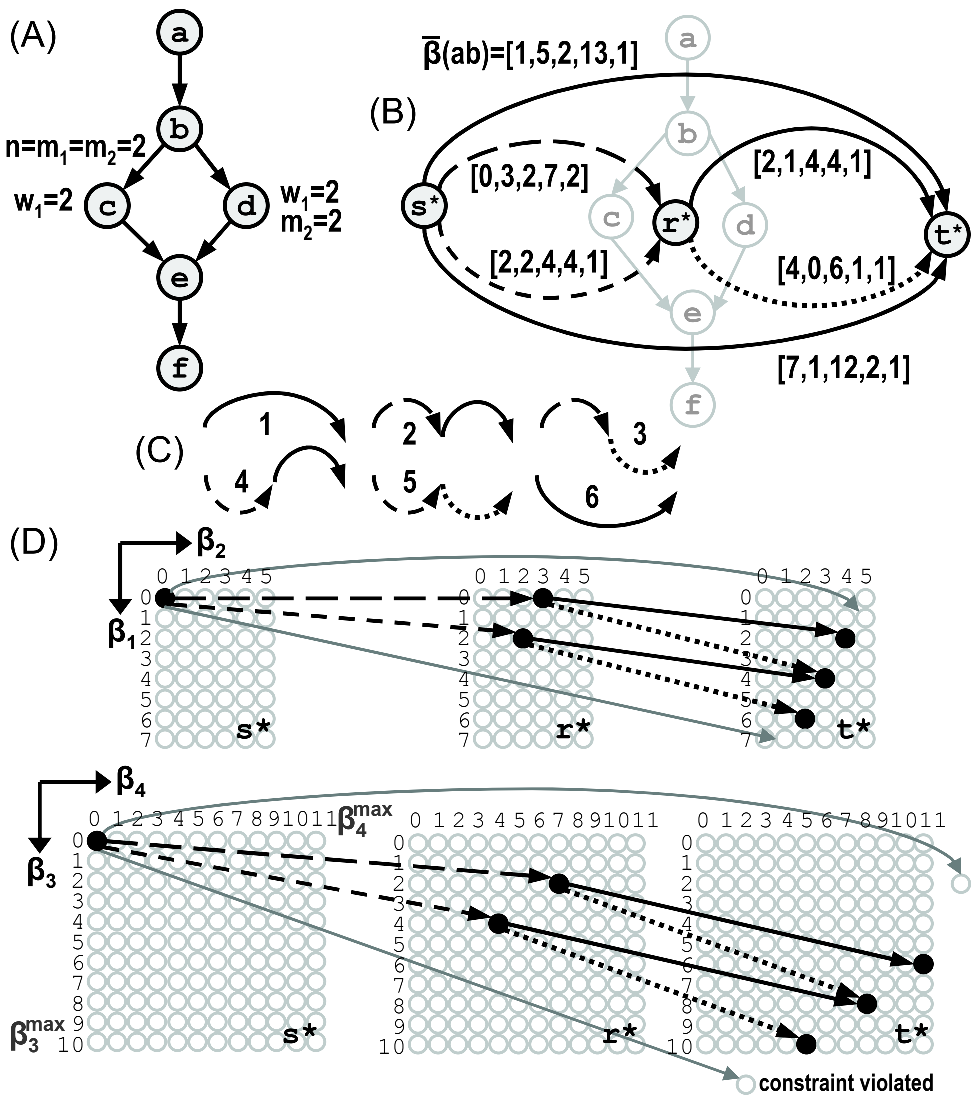


FIGURE 3.7. A) In task graph $G(a, f)$, all the attributes w_1, w_2, n, m_1 and m_2 are equal to 1, except for those shown in the figure. B) Dual graph $G^*(s^*, t^*)$ and the $\vec{\beta}$ attributes. C) Six possible paths to choose from. D) Construction of $G^\dagger$ in 4-D space.

$G^\dagger$ when:

$$\begin{cases} \text{there exists an edge } e^*(u^*, v^*) \text{ in } G^* \\ \text{and } \vec{i} + \vec{\beta}(e^*) = \vec{j} \end{cases} \tag{3.14}$$

where only the first four elements of $\vec{\beta}$ is used in this equation. In other words, for a given edge e^* connecting u^* to v^* in graph G^* and a given vector $\vec{i}$, there is a unique edge in $G^\dagger$ which connects vertex $u^*[\vec{i}]$ to $v^*[i + \vec{\beta}(e^*)]$. As a result, for a given path P^* from s^* to t^* in G^*, and a given start vector $\vec{o}$, there is a unique path in $G^\dagger$ starting from $s^*[\vec{o}]$.

That is, $\beta_5(u^*[\vec{i}], v^*[\vec{j}]) = \beta_5(u^*, v^*)$. Moreover, if we choose $\vec{o} = [0, 0, 0, 0]$, the path in $G^\dagger$ always ends at $t^*[\vec{\beta}(P^*)]$, because each edge from $u^*[\vec{i}]$ to $v^*[\vec{j}]$ in the path incurs an increase of $\vec{j} - \vec{i}$ in the value of $\vec{\beta}(P^*)$. In graph $G^\dagger$, graph structure replaces the attribute information except for β_5, which annotates the edges of $G^\dagger$ in the same way as edges of G^*. For example, consider cut $C_4 = \{ce, bd\}$ or equivalently $P_4^* = \{\text{dash}, \text{black}\}$. Using graph G^* (Figure 3.7.A), we calculate $\vec{\beta}(P_4^*)$ as $[2, 2, 4, 4, 1] + [2, 1, 4, 4, 1] = [4, 3, 8, 8, 2]$, while in graph $G^\dagger$ we have the same value (the first four elements of the vector) by simply looking up the index of t^* at the end of path P_4^* (Figure 3.7.D).

Therefore, one can evaluate Equation 3.13 for all possible paths by enumerating the vertices t^* at the end of every path P^* in graph $G^\dagger$. Let us denote the end point as $\tilde{t}(P^*)$. For example, $\tilde{t}(P_4^*) = [4, 3, 8, 8]$, because P_4^* ends at vertex $t^*[4, 3, 8, 8]$. The problem can be reformulated as:

$$\begin{aligned} &a) \text{ minimize:} \quad Q(P^*) = F\big(\tilde{t}_1(P^*), \tilde{t}_2(P^*), \|\beta_5(P^*)\|\big) \\ &b) \text{ constraints: } \tilde{t}_3(P^*) \le \beta_3^{\max} \text{ and } \tilde{t}_4(P^*) \le \beta_4^{\max} \end{aligned} \tag{3.15}$$

Note that constructing of $G^\dagger$ does not reduce algorithmic complexity of the partitioning. It is merely outlined for better visualization, and to enable subsequent optimizations.

3.3.3. Hard Constraints. A path in $G^\dagger$ that visits the vertex $v[\vec{i}]$ will lead to a partitioning solution whose first four attributes are at least as large as the elements of i. Therefore, we can trim out infeasible solutions, violating at least one of the hard constraints, during construction of $G^\dagger$. This is achieved not by trimming paths that violate Equation 3.15.b after construction of $G^\dagger$, but by refusing to insert violating edges in $G^\dagger$ in the first place. Each time we are about to connect an existing vertex $u^*[\vec{i}]$ to a new vertex $v^*[\vec{j}]$, violations of the hard constraints can be detected, in which case, vertex v and the edge are discarded. For example in Figure 3.7, assume $\beta_3^{\max} = 10$ and $\beta_4^{\max} = 11$. Path P_1^* violates the constraint because $0 + \beta_4(ab) \ge \beta_4^{\max}$, i.e., the 4th attribute of the new vertex $t^*[1, 5, 2, \underline{13}]$ is beyond the limit. P_6^* violates the constraint too. Thus, vertex $u^*[\vec{i}]$ is connected to vertex $v^*[\vec{j}]$ only if:

$$\begin{cases} \text{there exists an edge } e^*(u^*, v^*) \text{ in } G^* \\ \text{and } \vec{i} + \vec{\beta}(e^*) = \vec{j} \\ \text{and } j_3 \leq \beta_3^{\max} \text{ and } j_4 \leq \beta_4^{\max} \end{cases} \tag{3.16}$$

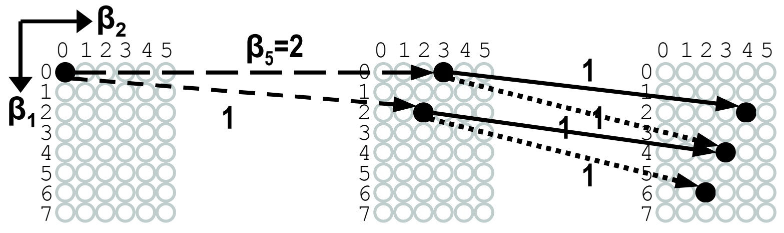
As a result, graph $G^\dagger$ is constructed such that any path from $s*$ to a t^* vertex in $G^\dagger$ identifies a feasible solution. It follows that Equation 3.15.b can be removed from consideration, and fewer paths that are left in $G^\dagger$ should be evaluated to optimize Equation 3.15.a.

Other constraints can be encoded similarly. For example, one could embed hard constraints on the maximum workload that could be assigned to either processor, which translates to trimming edges in $G^\dagger$ based on the first or second attribute.

3.3.4. Cost Function Minimization. The first four elements in the attribute vector of a path from $s^*[\vec{0}]$ to $\tilde{t}$ can be expressed based on the coordinates of the $\tilde{t}$ vertex in 4-D space. Hence, all paths that terminate at the same end point in $G^\dagger$ share the same attribute vector. That is, all such paths lead to task assignments that incur the same workload and memory distribution, and are only different in their inter-processor communication volume.

Among such possible task assignments, one would typically prefer the solution that incurs the minimum communication. This is to say that the cost function Q is non-descending in the fifth element of the attribute vector, i.e., $\|\beta_5(P^*)\|$. Equivalently, among a group of competing paths terminating at the same end point, we are interested in the path with minimum β_5.

Recall that the edges of $G^\dagger$ are annotated with their β_5 attribute. We can treat edge annotations as distance labels in $G^\dagger$, and run single-source multiple-destination shortest path on the graph from source vertex $s^*[\vec{0}]$ to all possible $\tilde{t}$ end points. The shortest path procedure would prune many possible paths that do not minimize cost function Q. Specifically, for every possible end point $\tilde{t}$, it only leaves one path in the graph that terminates there. For example in Figure 3.7.D, two paths arrive at the same end point: $\tilde{t}(P_3^*) = \tilde{t}(P_4^*) = [4, 3, 8, 8]$. However, only P_4^* is maintained for end point $t^*[4, 3, 8, 8]$ (Figure 3.8).



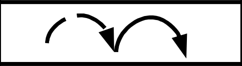
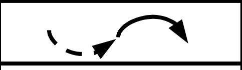
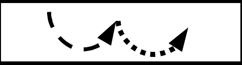
t1,t2	shortest path for this t	Q = max(W1+N, N+W2)	best path
2,4		max(2+3, 3+4) = 7	
4,3		max(4+2, 2+3) = 6	✓
6,2		max(6+2, 2+2) = 8	

FIGURE 3.8. Finding the shortest paths to t^* end points, and evaluating the cost function $Q = \max\{\|w_1(V_1)\| + \|n(c)\|, \|w_2(V_2)\| + \|n(c)\|\}$ for them.

We can calculate the value of Q at each of the $\tilde{t}$ points to find the end point, and the corresponding path, which globally minimizes Q (Figure 3.8). This procedure delivers an exact solution, which is guaranteed to optimize the cost function subject to hard constraints.

One might be able to prune out more candidate points from consideration, if more specific information about the cost function is known. For example, if one wants to avoid highly skewed workload distribution, he can ignore the end points whose w_1 or w_2 attribute is below or above a certain threshold.

3.3.5. Algorithm Complexity. The task graph and subsequent graphs constructed from that are directed acyclic graphs. Hence, the complexity of discussed transformations and the shortest path algorithm grow linearly with the number of edges in the subject graphs. Note that on a DAG, single source shortest path can be implemented using topological sort and thus has linear complexity [**CLRS01a**].

In planar graphs, the number of edges grows linearly with the number of vertices. Therefore, time complexity of our algorithm is determined by the number of vertices in the largest subject graph, i.e., $G^\dagger$. There are at most $|V| \times \beta_1^{\max} \times \beta_2^{\max} \times \beta_3^{\max} \times \beta_4^{\max}$ vertices in $G^\dagger$, where $|V|$ is the number of vertices in the application task graph. Thus, our algorithm has the time complexity of $O(|V| \times \Pi_{i=1}^{4} \beta_i^{\max})$.

This is considered pseudo-polynomial because the terms $\beta_i^{\max}$ merely depend on the properties of the application graph and target platform. $\beta_3^{\max}$ and $\beta_4^{\max}$ represent the available memory on the processors. Moreover, $\beta_1^{\max}$ is independent of the path choice, because $\|\beta_1(P^*)\| = \|\beta_1(C)\| = \sum_{v \in G_1} w_1(v)$ and thus its maximum value, $\beta_1^{\max}$, is the computation workload of the entire application when all tasks are assigned to processor 1. A similar argument holds for $\beta_2^{\max}$.

Note that since our graph bi-partitioning problem is NP-Complete no algorithm with strictly-polynomial complexity exists unless $P = NP$. The pseudo-polynomial complexity does not impose a real constraint on practicality of our approach. Pseudo-polynomial time algorithms incur reasonable latency for typical problem instances, unless we have to deal with very large attribute numbers which are uncommon in many practical settings [**GJ90**]. In practice, the attributes can be normalized so that $\beta_i^{\max}$ values remain relatively small.

3.4. Approximation Method

In this section, we present an approximation method for our exact task assignment algorithm with strictly-polynomial complexity. The approximation algorithm takes as input an acceptable error bound ξ, e.g., 10%, and guarantees that quality, i.e., throughput, of the near-optimal solution would not be more than a factor of ξ away from the optimal throughput.

We reduce the complexity by simplifying graph $G^\dagger$ in the exact algorithm. We illustrate that partitioning quality is not degraded beyond the intended limit in the course of the process. Specifically, the number of vertices in the expanded graph $G^\dagger$ is reduced to $O(|V| \times \Pi_{i=1}^{4} \log \beta_i^{\max})$ from the original $O(|V| \times \Pi_{i=1}^{4} \beta_i^{\max})$. In each of the four β_i dimensions of the 4-D space in $G^\dagger$, we judiciously trim the $\beta_i^{\max}$ possible values of indices to only $\log \beta_i^{\max}$ distinct numbers.

The idea is to replace multiple attribute values with a single representative value for that attribute. That is, we would like to develop an approximation function $\ddot{\beta} = f(\beta)$ to generate a representative value $\ddot{\beta}$ for a range of β values. For a given number $\delta > 0$, the approximation function is defined as

$$\ddot{\beta} = f(\beta) = (1+\delta)^{\left\lfloor \log_{1+\delta}^{\beta} \right\rfloor} \text{ and } f(0) = 0 \tag{3.17}$$

We apply this function whenever a new edge is to be added to graph $G^\dagger$. Thus, for constructing the path corresponding to P^* from s^* to t^* in G^*, the approximation function

(A)

β_i	0	1	$2-3$	$4-7$	$8-15$
$\ddot{\beta}_i = f(\beta_i)$	0	1	2	4	8

(B)

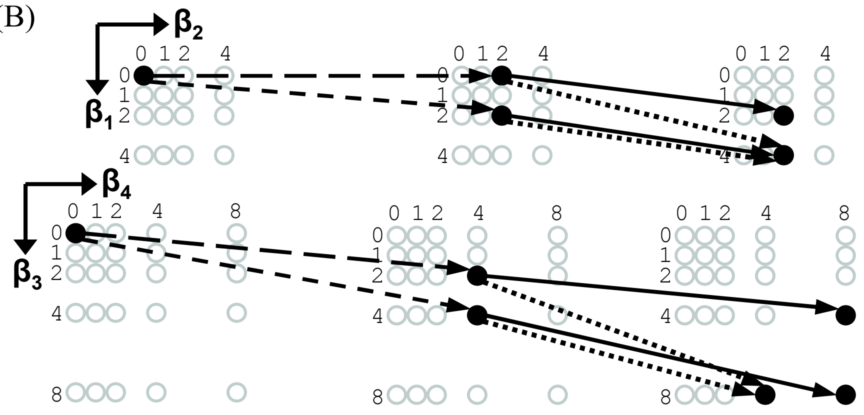


FIGURE 3.9. A) Approximation function. B) Constructing the approximated graph $G^\dagger$ from Figure 3.7.

is applied k times, where k is the number of edges in path P^*. Figure 3.9 illustrates the idea when $\delta = \vec{1}$, and possible values are trimmed to only 0, 1, 2, 4, and 8.

For example, consider path $P_3^* = \{\text{longdash}, \text{dot}\}$ in Figure 3.9. Originally, the longdash-marked edge connected $s^*[0,0,0,0]$ to $r^*[0{+}0, 0{+}3, 0{+}2, 0{+}7] = r^*[0,3,2,7]$. After applying the approximation, the path ends at $r^*[f(0+0), f(0+3), f(0+2), f(0+7)] = r^*[0,2,2,4]$. Similar procedure is applied to the next edge in the path, i.e., the dot-marked edge. It starts from the approximated vertex $r^*[0,2,2,4]$ and ends in $t^*[f(0{+}4), f(2{+}0), f(2{+}6), f(4{+}1)] = t^*[4,2,8,4]$.

After approximation, the number of vertices in the graph $G^\dagger$ becomes proportional to $|V| \times \Pi_{i=1}^{4} \log \beta_i^{\max}$, because in each of the four β_i dimensions, the above approximation function results in one of the following possible distinct numbers: $0, 1, 1{+}\delta, (1{+}\delta)^2, \cdots, (1{+}\delta)^b$, where $b = \left\lfloor \log_{1+\delta}^{\beta_i^{\max}} \right\rfloor$.

LEMMA 3.5. *We have* $\dfrac{\beta}{1+\delta} < \ddot{\beta} \leq \beta$

THEOREM 3.6. *If we set* $\delta = \sqrt[F]{1+\epsilon} - 1$, *where* $\epsilon > 0$ *and* F *is the number of faces in task graph* G, *then we have*

$$\frac{\|\beta(P^*)\|}{1+\epsilon} < \|\ddot{\beta}(P^*)\| \leq \|\beta(P^*)\| \tag{3.18}$$

This means that for every path P^* from s^* to t^* in the expanded graph $G^\dagger$, the approximated value $\|\ddot{\beta}(P^*)\|$ is at most degraded by a factor of $1+\epsilon$ from its original value $\|\beta(P^*)\|$.

Note that β and $\ddot{\beta}$ are vectors. In fact, δ is also a vector with four elements, which implies that we can have different approximation factors for different dimensions of the 4-D space. All mathematical operations are performed on each of the four dimensions independently.

The following theorems quantify the impact of approximation on memory constraints and cost function minimization.

Corollary 3.7. *The original hard constraints*

$$\|\beta_3(P^*)\| \le \beta_3^{\max} \text{ and } \|\beta_4(P^*)\| \le \beta_4^{\max} \tag{3.19}$$

can be replaced with following constraints, which use approximated values

$$\|\ddot{\beta}_3(P^*)\| \le \frac{\beta_3^{\max}}{1+\epsilon} \text{ and } \|\ddot{\beta}_4(P^*)\| \le \frac{\beta_4^{\max}}{1+\epsilon} \tag{3.20}$$

Corollary 3.8. *Let* $\ddot{\beta}^{\max} = f(\beta^{\max})$. *The constraints*

$$\|\ddot{\beta}_3(P^*)\| \le \ddot{\beta}_3^{\max} \text{ and } \|\ddot{\beta}_4(P^*)\| \le \ddot{\beta}_4^{\max} \tag{3.21}$$

guarantee that

$$\|\beta_3(P^*)\| \le (1+\epsilon)\beta_3^{\max} \text{ and } \|\beta_4(P^*)\| \le (1+\epsilon)\beta_4^{\max} \tag{3.22}$$

Therefore, to construct the new set of memory constraints, we may use either of the above two corollaries. The formula in Corollary 3.7 guarantees that the original constraints are met, at the expense of trimming some possibly-valid paths. The formula in Corollary 3.8 provides a new constant bound within a factor of $1+\epsilon$ from the bound in the original constraint. We proceed to consider the impact of approximation on cost function minimization.

Theorem 3.9. *Let* $\ddot{Q}(P^*) = F\big(\|\ddot{\beta}_1(P^*)\|, \|\ddot{\beta}_2(P^*)\|, \|\beta_5(P^*)\|\big)$ *denote the approximated value of our cost function* $Q(P^*) = F\big(\|\beta_1(P^*)\|, \|\beta_2(P^*)\|, \|\beta_5(P^*)\|\big)$, *for the path* P^*. *We* *have*

$$\big(1 - \frac{\epsilon}{1+\epsilon} S(P^*)\big)\ Q(P^*)\ \leq\ \ddot{Q}(P^*)\ \leq\ Q(P^*) \tag{3.23}$$

where $S(P^*)$ is defined as

$$S(P^*) = \frac{\|\beta_1(P^*)\|}{Q(P^*)} \max \frac{\partial Q}{\partial \beta_1} + \frac{\|\beta_2(P^*)\|}{Q(P^*)} \max \frac{\partial Q}{\partial \beta_2} \tag{3.24}$$

Note that although Q is originally a discrete function, throughout the following mathematical analysis, we look at it as a continuous function. That is, we use the same formula for Q, but assume its domain is $\mathbb{R}$ instead of $\mathbb{N}$. Note that Q does not have to be differentiable. As long as Q is differentiable on several intervals and continuous (i.e., piece-wise differentiable), we are able to calculate the maximum slope. For example, $\max \frac{\partial Q}{\partial \beta_1} = \max \frac{\partial Q}{\partial \beta_1} = 1$ for $Q = \max\{\|w_1(V_1)\| + \alpha_1\|n(c)\|, \alpha_2\|n(c)\| + \|w_2(V_2)\|\} = \max\{\|\beta_1(C)\| + \alpha_1\|\beta_5(C)\|, \alpha_2\|\beta_5(C)\| + \|\beta_2(C)\|\}$, because the slope of Q with respect to both $\|\beta_1(C)\|$ and $\|\beta_2(C)\|$ is either 0 or 1 on its entire domain.

Corollary 3.10. *Let $S^{\max}$ be the maximum possible value of S over the domain of function Q. We have*

$$\forall P^* : (1 - \frac{\epsilon}{1+\epsilon} S^{\max})\ Q(P^*) \leq \ddot{Q}(P^*) \leq Q(P^*) \tag{3.25}$$

The above theorem states that the error in calculating cost function is bounded within a constant factor. The main objective of graph bi-partitioning is to find the optimal path $P^* = P^*_{opt}$ which minimizes our cost function $Q(P^*)$. Using the approximation method, however, $\ddot{Q}(P^*)$ is minimized for some near-optimum path $P^* = P^*_{near}$.

Corollary 3.11. *Let $\xi = \frac{\epsilon}{1+\epsilon} S^{\max}$, and $T = \frac{1}{Q}$ denote the throughput. We have*

$$(1 - \xi)\ T_{opt} \leq T_{near} \leq T_{opt} \tag{3.26}$$

Consequently, for a given tolerable error bound ξ, throughput of the near-optimal solution would not be more than a factor of ξ away from the optimal throughput. The appropriate approximation factor δ is calculated using ξ in the following manner. First, we calculate ϵ from equation $\xi = \frac{\epsilon}{1+\epsilon} S^{\max}$, and then, δ from equation $\delta = \sqrt[F]{1+\epsilon} - 1$. The approximation process provides a controllable knob to trade task assignment quality with computation complexity and runtime.

3.5. Practical Extensions

3.5.1. Hardware-Inspired Versatility and Extensibility. We presented our technique using a selected set of attributes and an unconstrained cost function. In this section, we discuss extensions to our proposed task assignment algorithm, and demonstrate its utility in handling a diverse set of hardware customization scenarios.

One aspect of versatility is the ability to model execution period as a general, rather arbitrary, function of processor workloads ($\|w_k(V_k)\|$) and communication traffic ($\|n(C)\|$). This provides a simple yet effective way to fine tune the process of software synthesis to better match a specific hardware architecture. As an example, consider the hardware in Figure 3.10, and assume that there is no cache. A reasonable choice of the cost function would be $Q = \max\{\frac{\|w_1(V_1)\| + \alpha_1\|n(C)\|}{150}, \frac{hop}{400}, \frac{\|w_2(V_2)\| + \alpha_2\|n(C)\|}{200}\}$, to account for the disparity in clock frequency, network bandwidth and its latency. The term *hop* accounts for the router speed, and varies based on router implementation and statistical patterns of the traffic from other possible units. This cost function accounts for the details of onchip network, as well as the heterogeneity in processors' micro-architecture and clock frequency.

The technique can also be extended to handle other attributes. For example in Figure 3.10, using the w_i and n attributes alone, it is not easy to account for the effect of L1 cache on processor 1. In this case, we can add an additional attribute $ls_1(v)$ to vertices of task graph G. The attribute $ls_1(v)$ models the dynamic count of load/store instructions, in one firing of the task corresponding to vertex v. Similar to $\|w_1(V_1)\|$, we can define $LS_1 = \sum_{v \in V_1} ls_1(v)$. Subsequently, execution period estimation cost function can be generalized to $Q = F(\|w_1(V_1)\|, \|w_2(V_2)\|, \|n(C)\|, LS_1)$, in order to consider the impact of the new attribute on application throughput. For example, one might use the function $Q = \max\{\frac{LS_1 \times mr \times mp + \|w_1(V_1)\| + \alpha_1\|n(C)\|}{150}, \frac{hop}{400}, \frac{\|w_2(V_2)\| + \alpha_2\|n(C)\|}{200}\}$ to model the impact of cache misses. The extra cycles in processor 1 is equal to $LS_1 \times mr \times mp$, where mr and mp are the approximated miss rate and miss penalty, respectively. Note that we compact all attributes into one vector. As a result, addition of a new attribute, such as $ls_1(v)$, adds a dimension to the attribute vector and the space of graph $G^\dagger$.

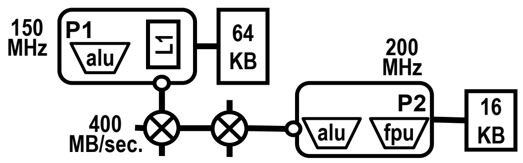


FIGURE 3.10. A sample heterogeneous dual processor hardware.

Similarly, we can remove extra attributes and simplify the formulation in case they are not relevant for a given hardware. For example if processors are homogeneous, we may eliminate attribute $w_2(v)$ and replace the term $\|w_2(V_2)\|$ in the performance estimation function with $W - \|w_1(V_1)\|$, where $W = \sum_{v \in G} w_1(v)$. In such cases, $\vec{\beta}$ would have fewer elements, and graph $G^\dagger$ would be constructed in a space with fewer dimensions.

We may also introduce or eliminate hardware-inspired hard constraints. For example, if we have a target hardware with separate data and instruction memory modules, we can introduce distinct constraints on the size of each memory module, e.g., $D_1 \leq$ data memory of processor 1, and $I_1 \leq$ instruction memory of processor 1. Similarly, one could introduce hard constraints on the processor workloads to eliminate solutions that result in highly skewed workload distributions.

3.5.2. K-Way Partitioning. We develop a heuristic based on the exact bi-partitioning method of Section 3.3 to partition the application task graph into arbitrary number (K) of subgraphs. The basic idea is to successively apply our bi-partitioning algorithm $K-1$ times to find $K-1$ cuts that partition the graph into K rather balanced pieces. Partitions of graph G with total workload of $\|w(V)\|$ each have an ideal workload of $\frac{\|w(V)\|}{K}$.

Figure 3.11 shows an example on $K = 4$ processors (note that K is not restricted to power of two). In this example, the exact bi-partitioning algorithm is first used to partition the task graph in two subgraphs (Figure 3.11.D), and then, each of them are further divided in two smaller subgraphs, one for each processor (Figure 3.11.E). As shown in Figure 3.11.B and 3.11.C, by recursively applying the exact bi-partitioning algorithm, we are able to assign tasks to both one- and two-dimensional pipeline platforms.

To apply the successive bi-partitioning, we calculate K_1 and K_2, which denote the number of processors available for tasks of each partition and hence, $K_1 + K_2 = K$. Our initial estimation is that K_1 is as close as possible to K_2, however, adjustments might be

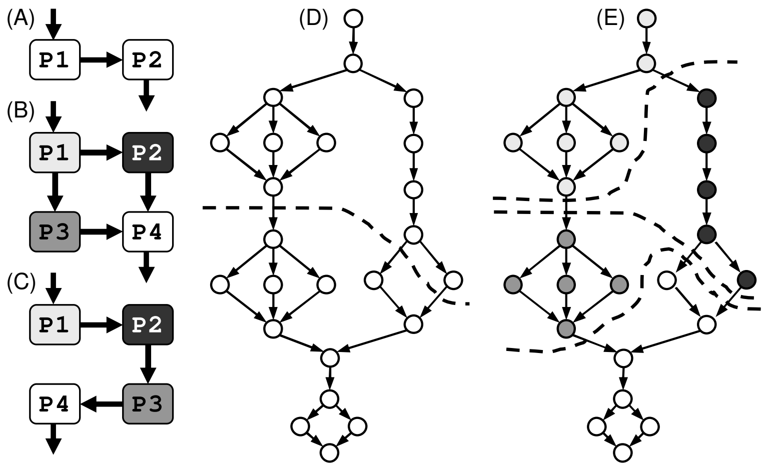


FIGURE 3.11. A) Pipeline dual-core. B) Two-dimensional pipeline quad-core. C) One-dimensional pipeline quad-core. D) Sample task assignment by exact bi-partitioning algorithm. E) Sample task assignment based on (D) for architectures in (B) and (C).

needed if the workload of the application cannot be equally distributed among the two sets of processors. For example if one of application tasks is much more intensive that the other tasks, the partition containing that task will have a much larger workload and hence, K_1 and K_2 have to be adjusted.

The bi-partitioning algorithm of Section 3.3 can be used in many ways, however, here we use it to develop only a one-dimensional K-way algorithm as follows. The cost function $Q = \max\{\|w(V_1)\| + \|n(C)\|, \|w(V_2)\| + \|n(C)\|\}$ is replaced with $Q = \max\{\frac{\|w(V_1)\|}{K_1} + \|n(C)\|, \frac{\|w(V_2)\|}{K_2} + \|n(C)\|\}$, where $K_1 = 1$ and $K_2 = 1$ in the original $K = 2$ case. Therefore, the cost function remains intact for bi-partitioning case.

In $K > 2$ cases, we try to consider the effect of future partitions. As the graph G is partitioned into two subgraphs G_1 and G_2, we take into account that G_1 and G_2 will be partitioned into K_1 and K_2 subgraphs, respectively. That's why in the new cost function, $\|w(V_1)\|$ is divided by K_1 and $\|w(V_2)\|$ by K_2. However, the communication overhead ($\|n(C)\|$) term is not changed in the updated cost function, because it should represent the cost of communication between processor K_1 and $K_1 + 1$ in the pipeline.

3.5.3. Task Graph Planarization. In our discussions so far, we assumed that application task graph is planar. Some programming languages, such as StreamIt [**TKA02**], guarantee the planarity of specified applications. However, our proposed method does not

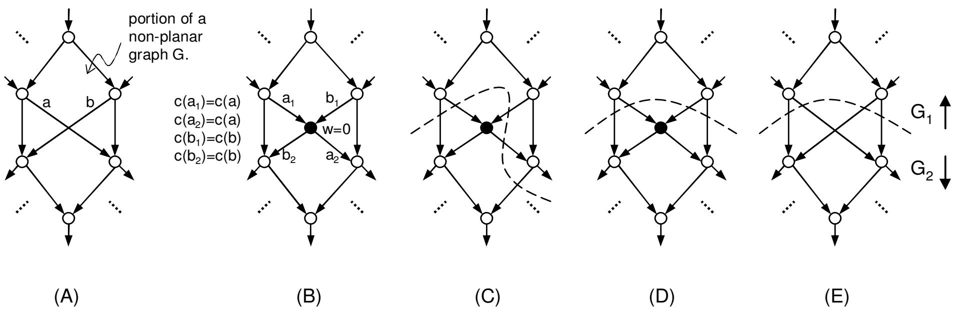


FIGURE 3.12. A) portion of an example non-planar graph G with one edge crossing. B) Transforming G into planar graph G_p by adding a dummy node C) An invalid cut in G_p. D) A valid cut in G_p. E) The resulting cut in G

require developers to use a specific programming language, and thus, the input graph might be non-planar. We introduce a transformation to *planarize* non-planar task graphs, and to make them amenable to our task assignment technique.

For a given non-planar input graph G, we start with the best embedding of G with minimum number of edge crossings. The goal is to eliminate edge crossings by adding dummy nodes at every edge crossing, while guaranteeing that graph partitioning estimations are accurate. We refer to the planarized graph as G_p. As an example, Figure 3.12.A shows a portion of a non-planar graph with one crossing and Figure 3.12.B is the planarized graph with one dummy node.

Conceptually, the dummy node passes data from its incoming edges to corresponding outgoing edges, i.e., data on edge a_1 goes to edge a_2 and the data on b_1 goes to b_2. However, the node and its corresponding computation do not appear in the generated code. We assign computation workload and communication cost values to the dummy bypass node and introduced edges, so the G_p can be correctly analyzed using our algorithm. We show that convex cuts of G_p correspond to convex task assignments in G.

The introduced edges in G_p have the same communication latencies as their corresponding edge in the original non-planar graph G. Specifically, $c(a_1) = c(a_2) = c(a)$ and $c(b_1) = c(b_2) = c(b)$ (Figure 3.12.B). In addition, workload of the dummy node is set to zero ($w(dummy) = 0$), because dummy node does not introduce any additional computation in the synthesized software. It will be removed before code generation by carefully reordering data communication during code generation.

Note that a convex cut in G_p gives a convex task assignment in G. Furthermore, the computation workload and communication latency of such a convex cut in G_p accurately models the workload and inter-processor communication of the corresponding task assignment in G. A convex cut in G_p does not cross both a_1 and a_2. Crossing both a_1 and a_2 implies non-convexity (Figures 3.12.C and 3.12.D), and is not considered in this chapter. As a result, we partition G_p without crossing the same edge of G twice, which enables us to infer a valid task assignment solution after partitioning G_p (Figure 3.12.E).

3.6. Empirical Evaluation

In this section, we first evaluate the dual-processor task assignment method (Section 3.3), and next, evaluate the k-way hueristic for a chain of processors (Section 3.5).

3.6.1. Soft Dual-Processor Platforms. Our evaluation is based on measurements of application throughput and memory requirement using operational binary on prototyped hardware. We use Altera DE2 FPGA board to implement soft dual-processor platforms using NiosII/f cores [**Nio**]. NiosII/f is a 32-bit, in-order, single issue, pipeline RISC processor with configurable architectural parameters.

We experiment with several different hardware configurations. In all cases, processors and the communication link run at 100 MHz. Processors use tightly coupled instruction memory constructed out of FPGA's onchip RAM blocks. Both processors have integer multipliers, but in processor p_1 it is built out of logic elements (LEs), and in p_2 it is built out of FPGA's embedded DSP blocks, which offers better performance. Figure 3.13 summarizes the configuration of the $2 \times 2 \times 3 = 12$ soft dual-processor platforms used in our study:

Computation: We experiment with two microarchitecture configurations, referred to as `A` and `B`. In setting `A`, p_2 has a hardware floating point unit (FPU), while p_1 uses software emulation to carry out floating point operation. In `B`, p_1 has FPU and p_2 uses software emulation.

Communication; Similarly, there are two different communication links (`F` or `R`). In setting `F`, inter-processor link is a 32-bit FIFO channel with 256 words buffer. We instantiate the link using Altera Onchip FIFO Memory in SOPC Builder software. In setting `R`, processors are connected through a third processor that emulates the function of a router in a network-on-chip (NoC). To model the impact of network traffic on message delivery latency in NoC, routing latency is assumed to be a random variable with normal distribution.

Memory: A total of 32 KB is available for instruction memory. We consider three different settings for allocating memory to processors: `08-24`, `16-16` and `24-08`. For example, the first row in Figure 3.13 refers to a configuration in which processor p_1 and p_2 have 8 KB and 24 KB of memory, respectively. The next two rows show two different distribution of memory space between the two processors.

Name	Processor p_1			Link	Processor p_2		
	int	float	imem		int	float	imem
`A-F-08-24`	LE	-	8K	fifo	DSP	FPU	24K
`A-F-16-16`			16K				16K
`A-F-24-08`			24K				8K
`B-F-08-24`	LE	FPU	8K	fifo	DSP	-	24K
`B-F-16-16`			16K				16K
`B-F-24-08`			24K				8K
`A-R-08-24`	LE	-	8K	router	DSP	FPU	24K
`A-R-16-16`			16K				16K
`A-R-24-08`			24K				8K
`B-R-08-24`	LE	FPU	8K	router	DSP	-	24K
`B-R-16-16`			16K				16K
`B-R-24-08`			24K				8K

FIGURE 3.13. Target soft dual-processor platforms.

3.6.2. Evaluation Methodology. We implemented our method in StreamIt 2.1 compilation framework [**G$^+$02**]. The compiler takes as input an application specified in StreamIt language, and after static scheduling and partitioning of the graph, generates separate `C` codes for execution on parallel processors. To generate executable binary, each `C` code is compiled with NiosII IDE C compiler (-O2 optimization) for its corresponding target Nios processor. Subsequently, applications' code sizes are obtained from the generated executable binaries. If the code size is larger than the available instruction memory, it is not possible to execute it on the soft processors. For feasible cases, the binaries are mapped to corresponding processors in prototyped architectures. Subsequently, application steady state throughput is measured during execution on FPGA.

We compare three different task assignment algorithms. First, we evaluate the proposed versatile algorithm described in this chapter, including the approximation method with parameter $\epsilon = 0.1$ for workload values. We also evaluate the algorithm without considering the heterogeneity in the architecture, i.e., $w_1(v) = w_2(v)$ for all tasks, and without considering the memory size constraints. Third, we use StreamIt built-in task assignment algorithm, which does not directly address heterogeneous architectures or memory constraints [**G$^+$02**]. We refer to the versatile algorithm as `Ver`, its simplified version as `Pre`, and StreamIt as `Str`.

`Pre` handles heterogeneous architectures only by estimating their relative performance with a constant factor r, which is established by profiling the processors using a set of representative applications. For example, we found that for AF and AR configurations,

$r = 0.7$ models the relative performance of processor p_1 over p_2, for benchmark applications containing floating point operations. Similarly, $r_{AF} = r_{AR} = 0.4$ is the relative performance for integer benchmarks. *Pre* does not consider memory constraints.

3.6.3. Attribute Estimation. Using high-level information available to the compiler, we estimate the attributes with simple models. Our empirical observations validate the effectiveness of the models.

Communication: As discussed previously on page 9 in Chapter 2, attribute $n(e)$ can be calculated by analyzing the SDF graph.

Computation: We profiled NiosII/f processors to estimate their cycle per instruction (CPI) distribution. Subsequently, internal computations of tasks are analyzed at high-level, and a rough mapping between high-level StreamIt language constructs and processor instructions is determined. For SDF-compliant streaming applications, control-flow characteristics are minimal. As a result, we employed first order estimations, such as average if-then-else path latencies, whenever needed. To account for the effect of input-dependant control flow, we consider the average latency of input-dependant tasks for few randomly selected input data. The analysis derived $latency_k(v)$, which represents the execution latency of one firing of task v on processor p_k. Therefore, as discussed previously on page 9, $w_k(v)$ is estimated as $r(v) \times latency_k(v)$.

Memory: Estimating the memory requirement is similar to the above workload estimation, except that here we count the number of assembly instructions corresponding to high-level statements of StreamIt language without actually compiling the applications to assembly. The analysis derived $m^*_k(v)$, which represents the amount of instructions memory needed for one firing of vertex v on processor p_k. Therefore, $m_k(v) = \text{mem}(r(v), for) + m^*_k(v)$, where $\text{mem}(r(v), for)$ is for the loop that would iteratively execute task v. Hence, if $r(v) = 1$, $\text{mem}(r(v), for) = 0$. Otherwise, it is a constant number, e.g., 12 in our platforms. Note that estimations are tuned to -O2 compiler optimization switch.

3.6.4. Performance Estimation and Memory Constraints. We mentioned that the cost function modeling application execution period should be tailored to target platform. In the first six target configurations, `A-F-` and `B-F-`, the communication link is a FIFO channel with single-cycle delivery latency. Therefore, we customize the cost function

$Q_{AF} = Q_{BF} = max(\|w_1(V_1)\| + \alpha_1\|n(C)\|, \|w_2(V_2)\| + \alpha_2\|n(C)\|)$, with $\alpha 1 = \alpha_2 = 8$. The parameter $\alpha 1$ (α_2) models the additional workload that has to be introduced on the sender (receiver) to write (read) a word of data to (from) the FIFO. Its constant value depends on our software generation setup. Similarly, in the next six targets, `A-R-` and `B-R-`, we use $Q_{AR} = Q_{BR} = max(\|w_1(V_1)\| + 8\|n(C)\|, \|n(C)\| \times avg(L), \|w_2(V_2)\| + 8\|n(C)\|)$, where $avg(L)$ is the average value of the random latency in the router.

In all cases, hard constraints are introduced to satisfy instruction memory constraints. There is no constraint on data memory, since it is large enough for all our benchmark applications. Hence, we only maintain attributes $m_i(v)$ (and not $m_i(e)$) because they represent code sizes. In case of `A-F-08-24`, for example, the set of constraints is $M_1(V_1) \leq 8$ KB and $M_2(V_2) \leq 24$ KB.

3.6.5. Benchmarks. Figure 3.14 shows different benchmarks used in our evaluations. They are well-known streaming applications that frequently appear in embedded application space. The applications are selected from the StreamIt benchmark set [**G$^+$02**], considering the memory constraints of our FPGA board.

3.6.6. Experiment Results. Figure 3.15 shows the accuracy of our high-level code size estimation scheme. The comparison and estimation accuracy are significant, because our task assignment algorithm uses the code size information to guarantee that generated binary code will fit in the limited instruction memory of soft processors. Note that we

Application	Description	Operations	$\|V\|$	$\|E\|$
`FFT`	Fast Fourier Transform	float	81	105
`TDE`	Time Domain Equalizer	float	50	60
`MMF`	Blocked Matrix Multiply	float	21	21
`MMI`	Blocked Matrix Multiply	int	21	21
`SORT`	Bitonic Sort	int (no mult.)	314	407

FIGURE 3.14. Benchmark applications. The last two columns show the number of vertices and edges in the application task graph.

	Processor	`FFT`	`TDE`	`MMF`	`MMI`	`SORT`
Estimate	p_1	2828	3944	3260	1920	7644
(bytes)	p_2	14384	13548	3740	2992	22832
Actual	p_1	2844	3864	3340	2320	5908
(bytes)	p_2	14560	12280	4196	3036	21296

FIGURE 3.15. Accuracy of high-level memory estimates for `A-F-08-24`.

decided to use estimates because at the time of task assignment the binary code is not available and is yet to be compiled. In addition, tasks are not *accurately* compile-able in isolation either, because of a variety of reasons such as volume-dependent communication, shared variable definitions, and the difference between inter-processor vs. intra-processor communication code.

On average, over all configurations and benchmarks, our high-level code size estimates are within 10% of the size of compiled binary. Some inaccuracy is inevitable at high-level, due to lack of information about many compilation and optimization decisions in generating binary code.

Figure 3.16 presents measured application throughput, normalized with respect to a single processor with a LE multiplier and without an FPU. The experiments highlight that the versatile algorithm (`Ver`) always produces code that fits in limited instruction memory

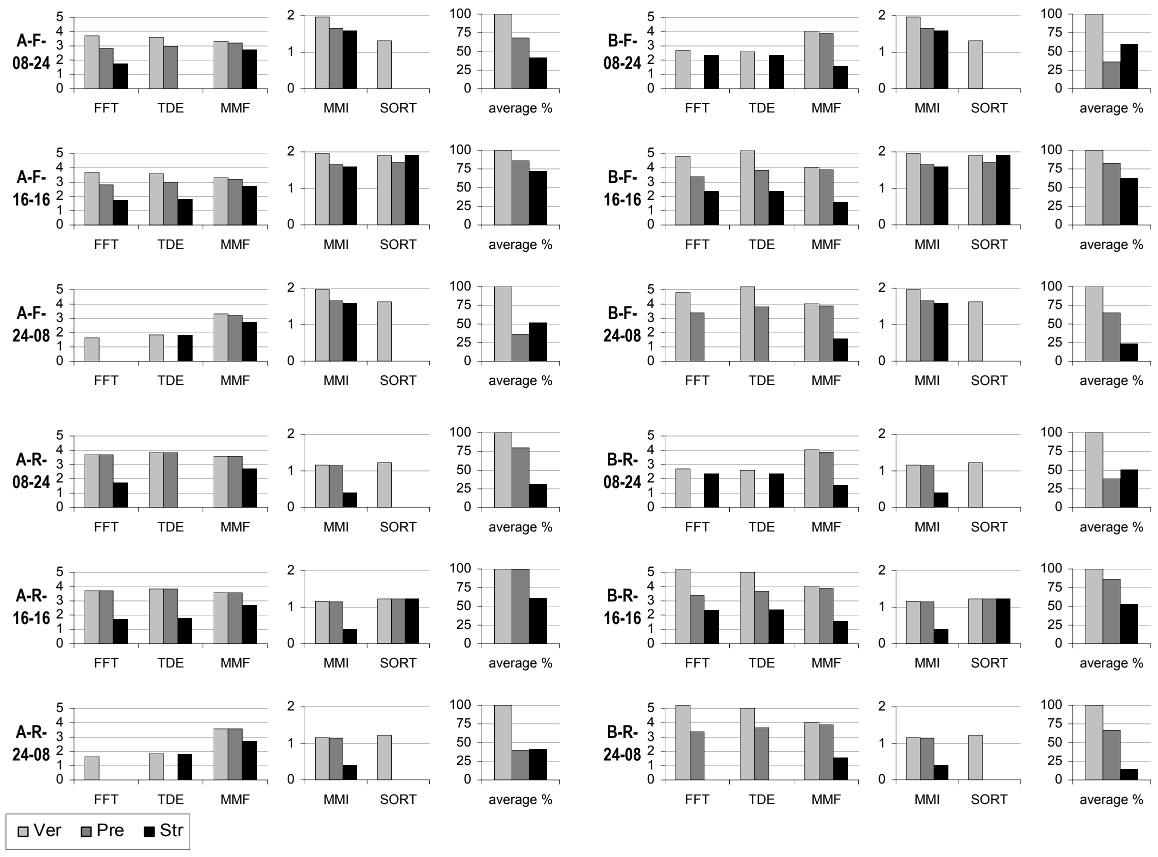


FIGURE 3.16. Applications' throughput normalized with respect to a processor with LE multiplier and no FPU. `Pre` and `Str` failed to generate feasible implementations for some architectures.

of soft processors. While for `-08-24` and `-24-08` configurations, the other two algorithms (`Pre` and `Str`) fail to generate feasible implementations for some benchmarks. For example in case of `SORT` application, which needs about 32KB memory, they always fail. `MMI` and `MMF` require small amount of memory, and therefore, they are feasibly implemented with all the three algorithms.

In addition, `Ver` consistently outperforms the other two techniques, and delivers the highest throughput in all cases. This underscores versatility of our proposed technique, and the fact that it can effectively handle heterogeneity and specific requirements of different target architectures.

Both `Pre` and `Ver` algorithms do not explicitly consider the impact of task assignment on code size, and therefore, fail to generate feasible solutions in all cases. For architectures with `-16-16` suffix, which have a balanced distribution of memory, all three algorithms produce feasible implementations.

`SORT` application contains a sequence of integer comparisons and conditional swaps, and no multiplications or floating point operations. Thus, `SORT` is indifferent to the micro-architecture heterogeneity that exists in our platforms. That is why `Str`, which assumes homogeneous processors, delivers the same throughput as `Ver`. On the other hand, `Pre` considers an average 0.7 factor in relative performance, which is irrelevant for `SORT`, and hence, degrades throughput.

Hardware FPU is substantially more efficient than software emulation of floating point operations. `MMF`, `TDE` and `FFT` have large numbers of such operations, and hence, their best throughput is observed when we assign as many tasks as possible to the processor with hardware FPU. Our integrated memory size estimation and heterogeneity modeling enables us to considerably outperform the competitors for floating point applications. In general, considering the effect of heterogeneous architecture is essential for most of the applications. Here, we can clearly see the advantage of `Ver` over the other two algorithms. `Pre` only partially considers platforms' heterogeneity, and often outperforms `Str`.

Figure 3.17 shows the tradeoff in execution time and memory usage of `Ver`, measured on a Linux/P4 machine for `FFT` benchmark and `A-F-100-100` platform, with respect to approximation accuracy. As expected, both runtime and memory usage of the algorithm decrease with decrease in approximation accuracy, i.e., with larger ϵ. Effectively, ϵ gives

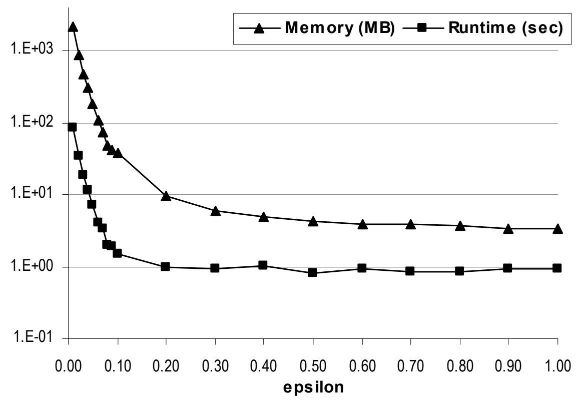


FIGURE 3.17. Memory usage and execution time of `Ver` for different approximation accuracy factors. Measured for `FFT` on `A-F-100-100` platform.

users a controllable knob by which, they can optimize algorithm's runtime and memory footprint in return for willingness to accept small deviation from optimal solution. For example in case of Figure 3.17, one could save about two orders of magnitude in runtime and memory usage by accepting at most 20% loss in quality, relative to the exact solution.

3.6.7. Experiment Results for K-way Hueristic. Based on the architecture AF in Figure 3.13, we constructed a chain of identical processors with up to 5 core that fit in our FPGA board. Figure 3.18 presents the measured throughput normalized by the single-core throughput. Results are shown for both our task assignment algorithms (called "Ver" in the figures) and StreamIt 2.1. Note that for $K = 2$, we use the optimal task assignment algorithm described in Section 3.3, but for $K \geq 3$ we use the heuristic approach of Section 3.5.

The following metrics and figures (3.19,3.20,3.21,3.22) present quantitative analysis for the above results. Figure 3.19 shows the amount of *extra* throughput gained by our method comparing to StreamIt 2.1:

$$100 \times \frac{Throughput - Throughput(StreamIt)}{Throughput(singlecore)} \tag{3.27}$$

For two processors, our method has 25%, 15% and 14% more throughput on `FFT`, `BSORT`, and `MATMUL`, and no extra gain on the other three benchmarks. This reconfirms our theoretical claim of optimality for exact task assignment algorithm of Section 3.3. The algorithm is always better than (or equal with) another method, which is StreamIt 2.1 in this case.

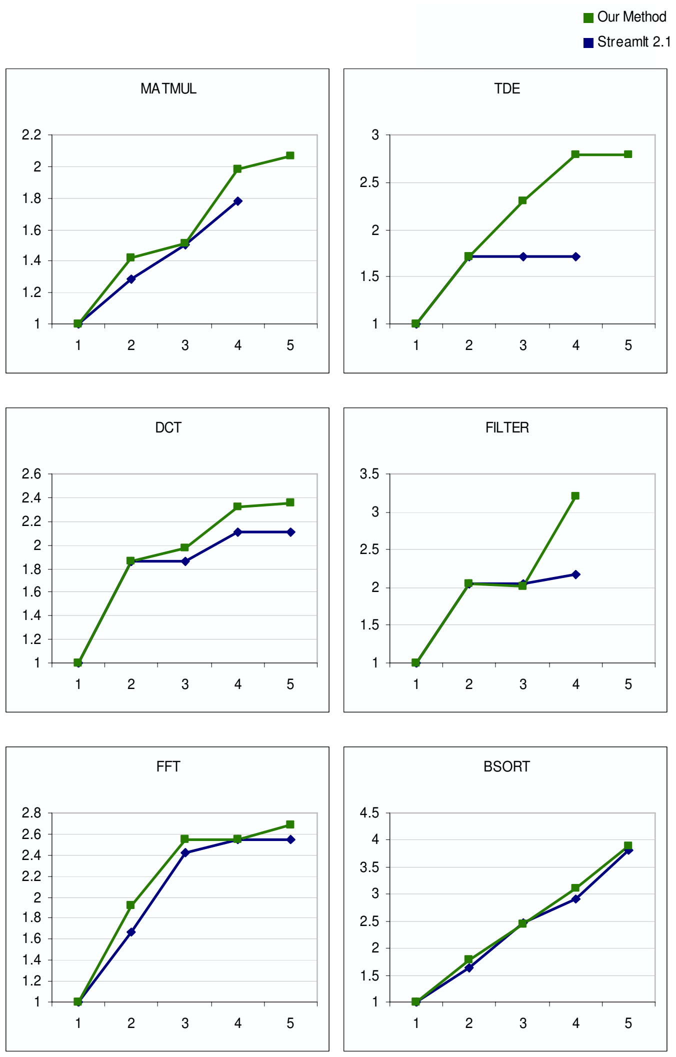


FIGURE 3.18. Normalized throughput for StreamIt 2.1 and our method. X axis is the number of processors and Y axis is the normalized throughput. We use the exact algorithm (Section 3.3) for $K = 2$, and the heuristic approach (Section 3.5) for $K \geq 3$.

	2-core	3-core	4-core	5-core
MATMUL	14	1	20	-
FFT	25	13	0	13
TDE	0	59	109	-
FILTER	0	-2	104	-
BSORT	15	-2	20	6
DCT	0	11	22	24
Avg.	9.0	13.3	45.8	14.3

FIGURE 3.19. Percentage of extra throughput gained by our method comparing to StreamIt 2.1, for $K = 2$, 3, 4 and 5 processors. Note that this data is *not* a comparison with baseline.

As shown in Figure 3.19 for $K \geq 3$, out of 18 different cases (3×6), the heuristic method gains an *extra* throughput of more than 100% on two cases. However, as shown in the figure, it may sometimes lead to throughput degredation. This is expected because there is no theoretically proven result on the optimality of heuristic methods in general.

Figures 3.20 and 3.21 show workload distribution of StreamIt 2.1 and our task assignment methods. For example, they show that StreamIt task assignment has assigned more than 50% of the total computation workload of `MATMUL` to fourth processor of the quad-core hardware ($p4$ in the figure). This is because `MATMUL` has a compute intensive node in its task graph which has a heavy workload.

The cost function mentioned in Section 3.3 does not directly minimize the workload imbalance, but as shown in Figure 3.20, our algorithm also balances the computation workload better than StreamIt 2.1. In some cases the hierarchical graph structure of the benchmark

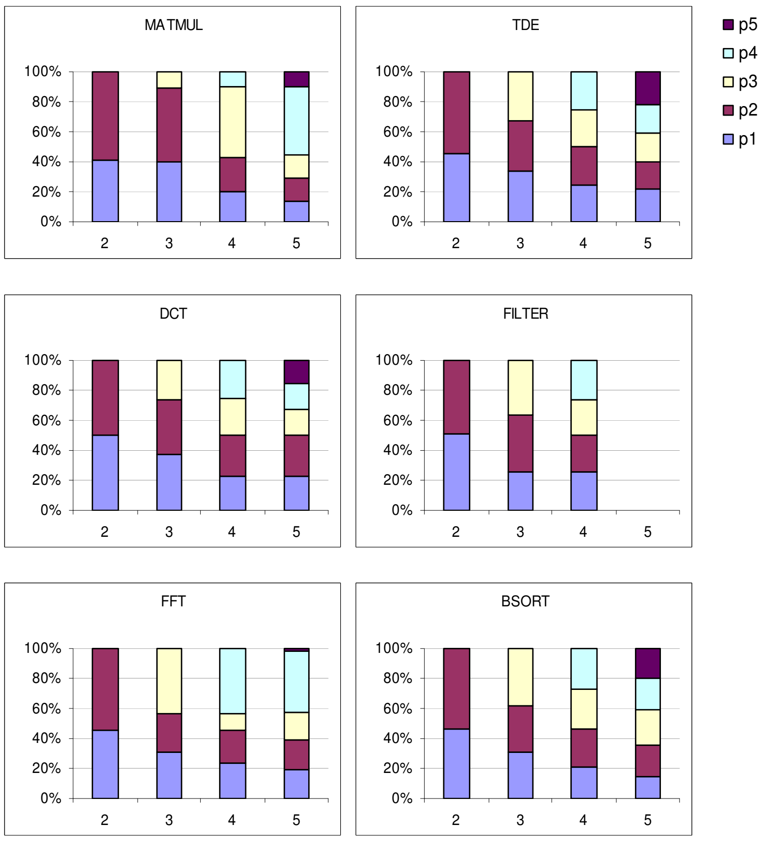


FIGURE 3.20. Workload distribution of our task assignment algorithm for K = 2, 3, 4 and 5 processors. X axis is the number of processors and Y axis is the workload distribution. Each color represents one processor of the multi-core hardware.

applications does not allow StreamIt 2.1 to effectively partition the task graph, but our graph bi-partitioning algorithm is not limited to the hierarchical structure, and therefore, has a larger search space. Figure 3.22 shows that this case happens for `TDE` application.

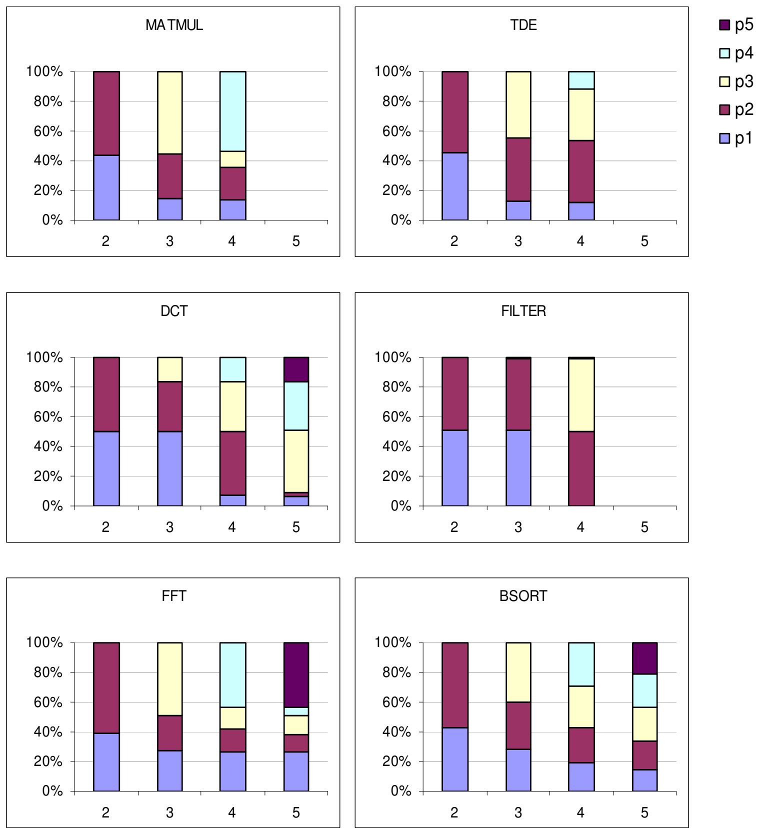


FIGURE 3.21. Workload distribution of StreamIt 2.1 task assignment algorithm for $K = 2$, 3, 4 and 5 processors. X axis is the number of processors and Y axis is the workload distribution. Each color represents one processor of the multi-core hardware.

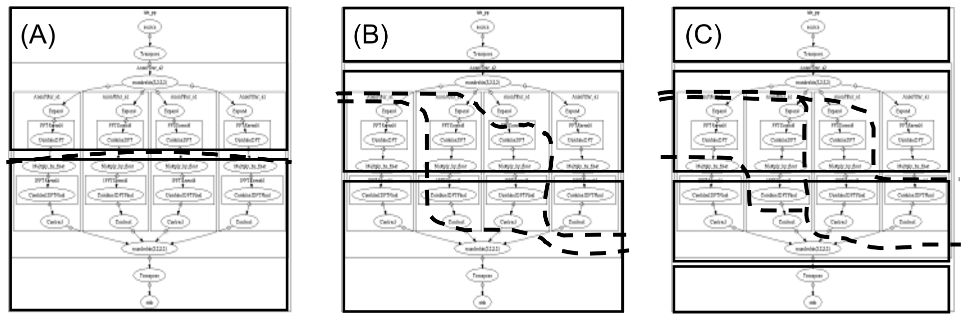


FIGURE 3.22. Task assignment for TDE with A) 2, B) 3 and C) 4 processors. Dotted lines show the results of our algorithm, and boxes show the results from StreamIt 2.1. A larger picture of the TDE application is available at *http : //leps.ece.ucdavis.edu/files/publication/matin/tde.png*

By comparing Figure 3.19 with 3.20 and 3.21, we notice that system throughput does not directly correlate to how well the workload is distributed. For example, in `MATMUL` with $K = 2$ cores, our algorithm increases the amount of workload imbalance but it actually achieves 14% higher throughput. This is mainly because the overall throughput is an implementation-dependent function of both computation workloads and communication overhead.

3.7. Related Work and Chapter Summary

A number of recent efforts address the task assignment problem for manycore software synthesis. Gordon et al. [**G$^+$02**] and Thies et al. [**TLA03**] describe a task assignment method to partially explore task parallelism for homogeneous platforms. Gordon et al. [**GTA06**] extend their work by a heuristic algorithm for acyclic StreamIt task graphs to exploit task, data and pipeline parallelism. As part of the Ptolemy project, Pino et al. [**P$^+$95**] propose a combined task assignment and scheduling method for acyclic SDF graphs on homogeneous platforms.

Stuijk et al. [**SBGC07**] propose a task assignment method for heterogeneous systems. Tasks are first sorted based on their impact on throughput, then a greedy method assigns one task at a time to the processor with the least workload. Cong et al. [**CHJ07**] present an algorithm for assignment of acyclic task graphs onto application specific soft multiprocessors. It starts by labeling the tasks, followed by clustering them into processors, and finally, tries to reduce the number of processors by packing more tasks onto under-utilized processors.

Some researchers have taken the approach of exploring the design space via enumeration. A multi-objective genetic task assignment algorithm for heterogeneous architectures is presented in [**EEP06**]. It considers power consumption, computation workload and communication overheads, however, it requires about 1000 generations to converge. Kudlur and Mahlke [**KM08**] propose an ILP formulation in order to deliver provably-optimal solutions. Approaches based on exhaustive search of the design space are typically not scalable because the runtime and/or memory requirement grow exponentially with problem size.

In this chapter we first presented a dual-processor task assignment algorithm that unlike other methods delivers provably-optimal solutions, with a reasonably low complexity by exploiting the planarity of target task graphs. Some SDFs, such as those specified in StreamIt, are inherently planar. In addition, we developed a planarization transformation to handle non-planar task graphs. The method supports exact handling of heterogeneous processors and memory size constraints and versatility in targeting various platform customizations. We also developd a heuristic method for pipelined execution on targets with an arbitrary number of processors. Future directions include applying the formal properties discussed in this chapter to design of a versatile manycore task assignment with optimal solution and low complexity.

CHAPTER 4

Scalable Estimation of Application Throughput on Manycore Platforms

Measuring performance (throughput) of a parallel application on the actual manycore hardware platform is not always possible especially for future versions of the platform with larger number of cores that has not yet been implemented. Overall system performance is determined not only by the processors that execute the parallel software modules but also by the onchip network which transfers the inter-processor messages between the modules. Therefore, performance measurement through execution of the parallel software on other platforms such as GPU does not yield to relevant information because the effect of onchip network is not modeled accurately.

One alternative is cycle-accurate Verilog simulation of the manycore system. In this method, the entire system including processor cores, memory units and inter-processor communication network is simulated with a commercial Verilog simulator such as Modelsim. A similar method is cycle-accurate instruction-set simulation (ISS) which models only the behavior of the hardware units. Note that to carry out the simulation, the parallel software modules should be compiled into binaries and loaded into the model.

ISS simulation or RTL/behavioral Verilog simulation of a manycore system provides accurate information but the unreasonably long runtime makes it impractical, especially for large number of cores. In this chapter, we present a performance *estimation* methodology, called SEAM, with acceptable accuracy, high scalability with respect to the number of processors, and high degree of freedom in targeting platforms with different underlying onchip network architectures. We accurately model the effect of onchip network through Verilog simulation, but abstract the local execution phases of every processor in order to speed up the simulation. Our experiments show that the estimation error is less than 15% [**HHG08**].

4.1. SEAM: Sequential Execution Abstraction Model

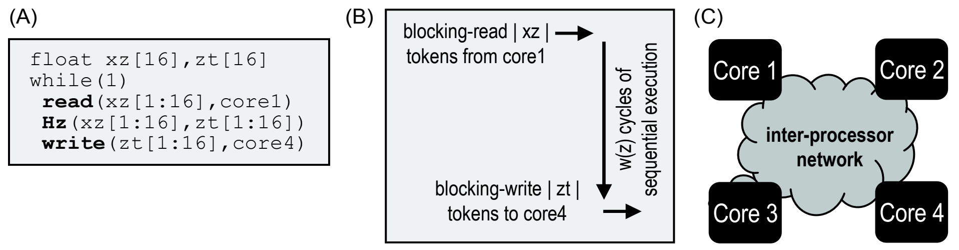
This section overviews our methodology in estimating throughput of a manycore system. The input to our performance estimation is an architecture model, including both processor and interconnect configuration, and associated task assignment and schedule of the application. The objective is to estimate throughput, without compiling the application.

4.1.1. Performance Model Reduction. Accurate manycore simulation requires cycle-accurate monitoring of instruction execution at each processor, complemented by cycle-accurate simulation of interconnect architecture, which is not scalable to larger number of cores. We introduce a simplified workload and communication characterization method that eliminates the need to cycle-accurately simulate instructions execution and data communication. Our technique preserves coarse-grain temporal behavior of the application, and is both fast and accurate.

Our approach is to reduce the complexity of performance model by coarsening temporal behavior of concurrent software modules. We view a software module (running on one of the processors) as a series of computation and communication phases, and hence, we attempt to characterize latency and volume of computation and communication phases at system-level. Subsequently, a substantially-reduced Verilog model is generated for the system in which, characterized temporal phases replace software module instruction sequence. Simulating the reduced model is considerably faster, while giving reasonable performance accuracy.

Figure 4.1 visualizes an example of our model simplification. A sample synthesized software is shown in Figure 4.1.A. Our technique views the processors as traffic generator/consumer with known delay elements, and simplifies temporal behavior of the code based on estimations (Figure 4.1.B). Details of our model reduction is the following.

4.1.2. Task Workload Characterization. The first step to estimate system performance, is to characterize the workload intensity associated with each task. Although compilers can perform various inter-task optimizations after several tasks are assigned to the same processor, it is reasonable to ignore such potential optimizations at system-level. Therefore, we analyze tasks in isolation and associate a deterministic number to each task that represents its workload latency on its allocated processor.


(A)
(B)
blocking-read | xz | tokens from core1
w(z) cycles of sequential execution
blocking-write | zt | tokens to core4
(C)
Core 1
Core 2
inter-processor network
Core 3
Core 4

```
float xz[16],zt[16]
while(1)
 read(xz[1:16],core1)
 Hz(xz[1:16],zt[1:16])
 write(zt[1:16],core4)
```

FIGURE 4.1. Example based on Figure 2.2 on page 8: A) Synthesized code for core #2. B) The simplified performance model for core #2. C) Performance estimation of the target manycore system based on the simplified model.

Specifically, we profile processors to estimate their cycle per instruction (CPI) distribution. Next, tasks internal computations are analyzed at high-level, without compiling to assembly, to derive a rough mapping between high-level language constructs and processor instructions. Furthermore, we use first order control-flow estimation such as average if-then-else path latencies, and expected number of loop iterations, to account for application control-flow behavior. Note that streaming applications are mostly data-flow intensive and their control-flow characteristics are minimal. This one-time analysis derives the estimated number of clock cycles needed for execution of a task on its assigned processor.

Similarly, inter-task communication cost is estimated from high-level application specification in which, parallelism is explicit. Given a specific task assignment and for a particular task, the number of data tokens that might have to be transmitted to other processors is readily calculated by checking the processors to which immediate descendant of the task are allocated. For applications modeled as synchronous dataflow graphs, each node appears a specific number of times in the steady state schedule. The number of appearances and data production and consumption rates are known statically, which enable quick one-time characterization of tasks data communication volume.

4.1.3. Processor Workload Characterization.

Tasks assigned to the same processor are scheduled to generate the corresponding executable code (Section 2.3.4). Scheduling impacts temporal behavior of code running on the processor, because temporal motion of inter-processor read/write operations might create or eliminate blockings on communication channels. Therefore, parallel application performance directly depends on task schedules, among other factors.

At system-level, the workload associated to a processor can be estimated to be a straightforward combination of workloads of tasks that are assigned to that processor, according to the given schedule. Note that this estimation incurs inaccuracies because combined workload of several tasks (in terms of cycles) is not necessarily equal to sum total of tasks workloads that are characterized in isolation. Compiler optimizations, such as enhanced register allocation to reduce register spilling, typically improve upon sum total of tasks workloads. However, “sum total” estimation is a good approximation at system-level.

Consequently, we simplify temporal behavior of a processor workload to a sequence of computation, with estimated latency, and communication, with estimated volume. This simplification is used to generate a temporal behavior model for each interconnect port, which replaces processor and its executable binary in a cycle-accurate simulation.

4.2. Empirical Evaluation

We utilized Xilinx Embedded Development Kit (EDK) to develop the aforementioned soft multi-processor synthesis framework. Xilinx EDK provides a library of soft processors and interconnect primitives. Xilinx 32-bit soft processor, called Microblaze, can be customized in a number of ways. We characterized Microblaze processor to determine its clock per instruction (CPI) distribution.

Microblazes can be interconnected using Xilinx Fast Simplex Links (FSL), which essentially implement point-to-point FIFO channels with configurable buffer size and width. We also implemented a standard network-on-chip router element with five bidirectional ports. Typically, one port connects to a processor and the remaining four ports are connected to neighboring routers. Hence, routers can be cascaded to interconnect an arbitrary-sized mesh of processors.

Figure 4.2 illustrates the flow of our experiments. We utilize StreamIt [**TKA02**] compiler to analyze and implement high-level application specifications. StreamIt is a programming language whose semantics are closely related to synchronous dataflow model of computation [**LM87a**] with a few enhancement. Specifically, standard SDF model is enhanced to allow application initialization phase and utilization of limited control flow in program specification. In addition, StreamIt provides an open-source compilation framework for stream programs specified in its language. StreamIt compiler takes as input an application

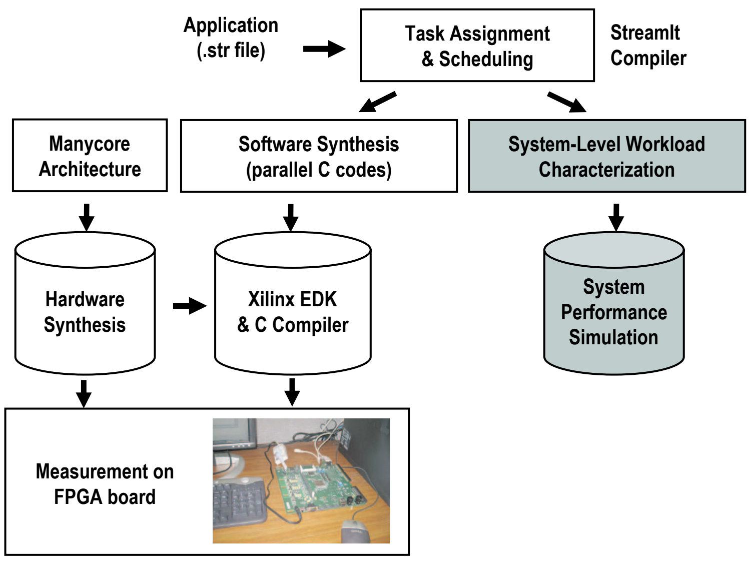


FIGURE 4.2. Experiments flow

specified in enhanced synchronous dataflow (SDF) semantics with StreamIt syntax, and after partitioning of the task graph, generates parallel `C` codes for parallel processors. Parallel codes should be compiled for the target uni-processor to generate executable binary.

We implemented our task and processor workload characterization method in StreamIt before generating `C` codes. Processor workloads are estimated using high-level specification of the applications in StreamIt language and Microblaze CPI distribution, as discussed in Section 4.1. In addition, StreamIt candidate task assignment and schedule are used during processors workload characterization.

The characterized workload is spelled out as a simple Verilog code that breaks up estimated temporal behavior of the processors into computation, with known latency, and communication, with known volume, phases (Section 4.1). This models is integrated with the interconnect architecture model, and simulated using Modelsim simulator.

In order to evaluate the accuracy of our results, we implemented candidate architectures on Xilinx ML310 board that has a Virtex II Pro FPGA. StreamIt and Microblaze `C` compiler (`mb-gcc`) were used to generate executable binaries from high-level application task graph. Application throughput measurements from operating hardware are used as baseline to determine the estimation accuracy.

Arch. name	# of processors	Interconnect arch.	Buffer Size
3x2-2	6	$3x2$ packet-switch	2
3x2-4	6	$3x2$ packet-switch	4
3x2-8	6	$3x2$ packet-switch	8
2x2-2	4	$2x2$ packet-switch	2
2x2-4	4	$2x2$ packet-switch	4
2x2-8	4	$2x2$ packet-switch	8
4-FSL-16	4	FSL cascade	16
4-FSL-32	4	FSL cascade	32
2-FSL-16	2	FSL cascade	16
4-FSL-32	2	FSL cascade	32

FIGURE 4.3. Candidate interconnect architectures and configurations

We designed ten different candidate architectures, which are summarized in Figure 4.3. Candidate architectures contain different number of Microblaze processors that are interconnected using either packet-switched on-chip network or point-to-point FIFO channels (FSL). The buffer size in both packet-switching routers and FIFO channels are configured to explore different design points.

We selected five representative streaming applications from StreamIt benchmarks: Bitonic Sort, FFT, Filter Bank, Blocked Matrix Multiplication, and Time Devision Equalization. We estimated system throughput for all applications on the aforementioned ten candidate architectures. Candidate architectures were also implemented on FPGA board and applications were mapped to measure performance.

Figure 4.4 illustrates the average runtime of SEAM in estimating the performance across all candidate architectures. The simulations were run for 100 iterations in order to reach steady state. The last entry on X-axis shows the geometric mean value for all applications.

As shown on the right vertical axis of Figure 4.4, the error in estimating the performance with SEAM is around 15%. This error is calculated as the relative difference between the actual throughput measured on FGPA, and estimated throughput using our model. The estimated throughput is consistenly lower than actual measurements, which suggests that our workload estimation is slightly pessimistic.

Inaccuracy in throughput estimation is primarily due to two reasons. First, during high-level workload characterization a *rough* mapping between task specification and processor instructions are determined, which naturally is not perfectly accurate. Second, some

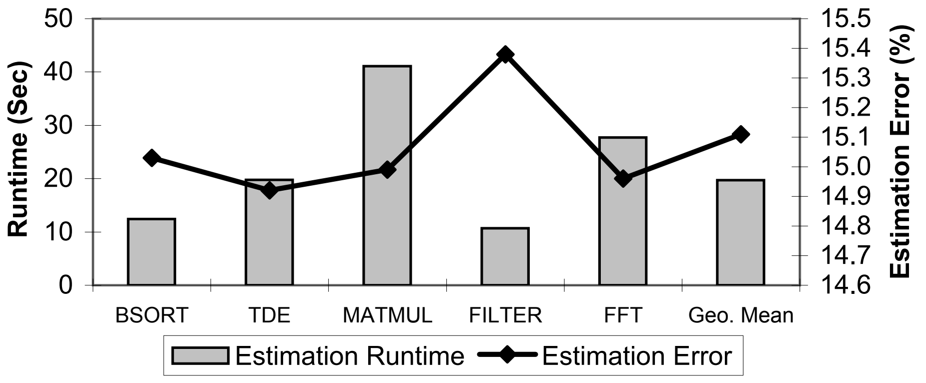


Figure 4.4. Performance estimation accuracy and runtime comparison.

compiler optimizations, such as improved register allocation, loop overhead reduction and memory access optimization, are enabled by combining multiple tasks that run on the same processor. At system-level, it is hard and runtime-expensive to consider such optimizations.

Later in the experiment section of Chapter 6 on page 105, we employed SEAM to estimate the performance of several applications and architectures. The workload estimates in that work are from profiling and hence, the effect of compiler optimizations are taken into account. The results show that SEAM is very accurate. We also considered the effect of cache size on the accuracy of SEAM in that chapter.

4.3. Related Work and Chapter Summary

Performance of a manycore system is determined not only by the processors that execute the parallel software modules but also by the onchip network which transfers the inter-processor messages between the modules. Therefore, performance measurement through execution of the parallel software on other parallel platforms such as GPU or multi-threaded CPUs does not yield to reliable performance measures because the effect of onchip network is not modeled accurately. Cycle-accurate Verilog or ISS simulations of the entire system [**LKJ$^+$08**] are unreasonably slow specially for manycore systems with large number of processors.

Stuijk et. al [**SGB08**] propose a performance estimation method, in which the application execution is essentially simulated at high-level using the task workload estimates. The simulation keeps track of the execution state, and finishes once steady state has been reached, at which point the steady state throughput is reported. They also propose a method to explore the tradeoff between application throughput and inter-processor buffer capacity.

SEAM employs the same idea in estimating the steady state throughput. The workload estimates are used to abstract the local execution phases of processors, and the application execution is simulated until steady state has been reached. However, SEAM advances the previous work in two ways. First, as opposed to considering a self-timed execution of all the tasks [**SGB08**], SEAM supports an arbitrary task assignment and task schedule. Second, through transformation to abstract Verilog models, SEAM quickly and accurately models the effect onchip network architecture. Currently, SEAM does not model the effect of cache because most embedded manycore [**KDH$^+$05, TCM$^+$08**] have local software managed memories. However, since stream programs present a high degree of cache locality, some architectures [**BEA$^+$08**] incorporate traditional caches, and hence, one direction for future enhancement of SEAM is to include high-level modeling of cache behavior without line by line simulation of the (binary) code.

CHAPTER 5

Throughput-Memory Tradeoff via Overlapping Application Iterations

Judicious scheduling of tasks in time is crucial in maximizing the throughput. The focus of traditional task scheduling techniques is to optimize a *periodic* schedule for tasks that are assigned to the same processor. This approach can severely degrade the throughput if task assignment tries to better balance the workload by allowing non-convex cuts, e.g., by grouping the tasks that are far away from each other in the SDF graph.

Iteration overlapping, which is closely related to software pipelining, schedules the tasks which are assigned to the same processor in time, while allowing multiple iterations of the periodic application execution be alive at the same time. This adds an initial execution phase before the periodic execution, and provides the opportunity to achieve a much higher throughput.

In this chapter, we investigate iteration overlapping as a controlling knob to trade application throughput with its code size. Given an initial task assignment and scheduling, our objective is to construct overlapped local task schedules that would correctly implement the application, and would lead to competitive throughput-memory tradeoff points. We rigorously analyze and present underlying properties of the formulated tradeoff. For example, we present the bounds on maximum throughput, minimum code size and constraints on valid overlapped schedules. We utilize the properties to develop an algorithm that generates a set of competitive design points with respect to one another, in the two dimensional throughput-memory plane. We implement the technique within our software synthesis framework, and evaluate it on a number of applications and several parallel architectures. The experimental results confirm the effectiveness of our theoretical modeling and approach [**HG11a**].

5.1. Preliminaries and Definitions

In this section, we briefly review the process of baseline software synthesis, i.e., no iteration overlapping, and also, define the formal terms which are used in later sections to explain the properties of iteration overlapping.

5.1.1. Baseline Software Synthesis. We consider a special case of SDF in which the producer and consumer tasks of every channel have the same port rates. As a result each task needs to execute only once in the periodic execution, i.e., $r(v) = 1$ for all tasks. Figures 5.1.A and 5.1.B illustrate a simple task graph and a platform with two processors ($P = 2$).

Task Assignment: As mentioned before, first tasks are assigned to the platform processors. In the example, tasks a, b and d are assigned to processor p_1, and task c to processor p_2 (Figure 5.1.C). We define cut channels $C \subset E$ as the channels whose producer and consumer tasks are assigned to different processors. In our example, $C = \{x, y, z\}$. Formally,

$$C = \{e(v,u) \in E \mid p(v) \neq p(u)\} \tag{5.1}$$

where $p(v)$ denotes the processor that is allocated to task v. The tokens on a cut channel $e(v,u) \in C$ should be transfered from processor $p(v)$ to $p(u)$ using the platform FIFO channels. This is modeled by adding new `write` and `read` tasks to the task graph. Let us denote such tasks as e^k, where k is the processor that the new task belongs to. For example, task x^1 denotes the `write` task which is added to processor p_1 due to the cut edge x (Figure 5.1.D). Formally

$$\begin{gathered} p : V \to \{1, 2, \ldots, P\} \\ p(v) = k \iff v \in V_k \end{gathered} \tag{5.2}$$

where V_k denotes the set of tasks assigned to processor p_k including the `write` and `read` tasks, and E_k the set of channels between tasks $v \in V_k$. Therefore

$$\bigcup_{k=1}^{P} V_k = V \text{ and } \bigcup_{k=1}^{P} E_k = E - C \tag{5.3}$$

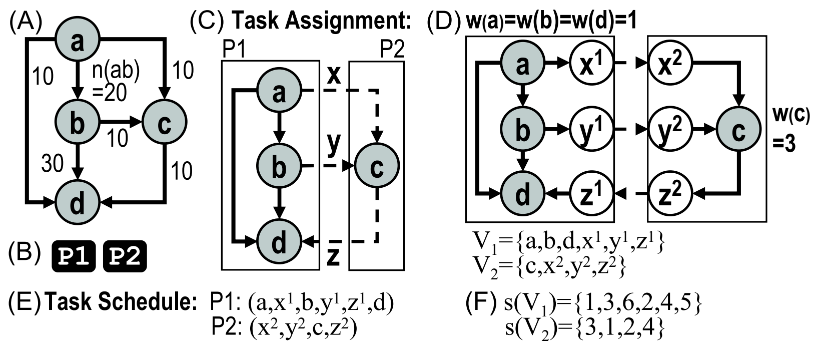


FIGURE 5.1. Example: A) Example task graph. B) Target dual processor platform. C) Tasks are assigned to processors. Cut channels are marked with dashes. D) New write and read tasks are added to the graph. E) Tasks are scheduled. F) Order function s represents the schedule.

Task Scheduling: Next, tasks that are assigned to the same processor are statically scheduled for infinite periodic execution on that processor. For example, a valid periodic schedule is (a, x^1, b, y^1, z^1, d) for processor p_1 and (x^2, y^2, c, z^2) for processor p_2 (Figure 5.1.E). Note that because of the inter-processor dependencies, i.e., the cut channels, scheduling of the tasks on one processor is not independent from scheduling of the tasks on other processors. For example, although (y^2, x^2, c, z^2) meets all the local dependencies in processor p_2, it is not a valid schedule because p_2 would read the wrong data from FIFO. The data written to FIFO by task x^1 is meant to be read by x^2 but in this invalid schedule it would be read by task y^2.

Formally, task schedule is defined as a total order $s(v)$ on the tasks v assigned to the same processor:

$$
\begin{aligned}
&s : V \to \mathbb{N} \\
&\forall v \in V_k : s(v) = s_k(v)
\end{aligned}
\tag{5.4}
$$

where

$$
\begin{aligned}
&s_k : V_k \to \{1, 2, \dots, |V_k|\} \\
&s_k(v) < s_k(u) \iff t(v_i) < t(u_i)
\end{aligned}
\tag{5.5}
$$

which holds if tasks v and u are assigned to the same processor, i.e., $p(v) = p(u) = k$. For example, $s(b) = s_1(b) = 3$ because b is the third task in the schedule of processor p_1 (Figure 5.1.F). In the above equation, $|V_k|$ denotes the number of tasks in V_k.

Throughout this chapter, for a task v, the term v_i denotes the i'th invocation (firing) of task v, i.e., the firing of task v from iteration i of the periodic application execution. We say v_i is the i'th instance of task v. The term $t(v_i)$ is the time at which i'th instance of task v is fired.

Since s_k is a one-to-one function, s_k^{-1} is defined and it refers to the task at a certain position in the schedule of processor p_k. For example, $s_1^{-1}(3) = b$. Formally

$$\begin{aligned} &s_k^{-1} : \{1, 2, \ldots, |V_k|\} \to V_k \\ &s_k^{-1}(j) = v \iff s_k(v) = j \end{aligned} \tag{5.6}$$

Task Firing Sequence: For a given task assignment and scheduling, the firing sequence of tasks in every processor is defined. We denote the *task firing sequence* in processor p_k with

$$\langle \Sigma_k \rangle_{i=0\to\infty} = \langle \sigma_{1,i}, \sigma_{2,i}, \ldots, \sigma_{|V_k|,i} \rangle_{i=0\to\infty} \tag{5.7}$$

where $\sigma_j = s_k^{-1}(j)$ is the task at position j in task schedule of processor p_k, and thus, $\sigma_{j,i}$ denotes[1] the i'th instance of task σ_j. As shown in Figure 5.2.A, the task firing sequence for processor p_1 in our example in Figure 5.1 is

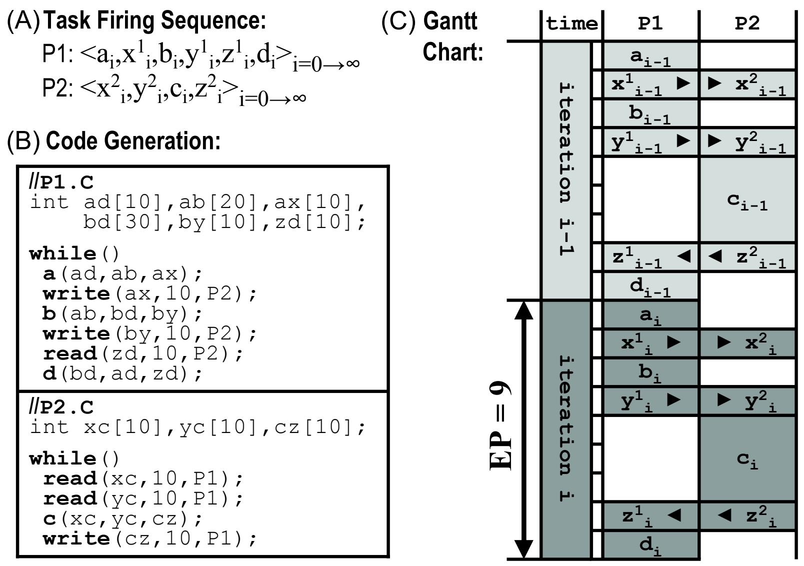
$$\langle \Sigma_1 \rangle_{i=0\to\infty} = \langle a_i, x_i^1, b_i, y_i^1, z_i^1, d_i \rangle_{i=0\to\infty} \tag{5.8}$$

whose unrolled representation is

$$a_0, x_0^1, b_0, y_0^1, z_0^1, d_0, a_1, x_1^1, b_1, y_1^1, z_1^1, d_1, \ldots \tag{5.9}$$

Code Generation: Given the sequence $\langle \Sigma_k \rangle_{i=0\to\infty}$, synthesizing the software code of a processor p_k is straightforward. A while loop fires the tasks iteratively in the order given by Σ_k. Figure 5.2.B shows the synthesized code for our example. Tasks read their inputs from, and write their outputs to arrays. For tasks that are assigned to the same processor, inter-task communication channels are implemented by passing the reference to the corresponding array. Inter-processor communication is implemented using the aforementioned `write` and

[1]Here the term $\sigma_{j,i}$ has two subscripts because the task name itself (σ_j) has a subscript. We had to use a second subscript, i, to denote the firing instance.



(A) **Task Firing Sequence:**

P1: $<a_i, x^1_i, b_i, y^1_i, z^1_i, d_i>_{i=0 \rightarrow \infty}$

P2: $<x^2_i, y^2_i, c_i, z^2_i>_{i=0 \rightarrow \infty}$

(B) **Code Generation:**

```
//P1.C
int ad[10],ab[20],ax[10],
    bd[30],by[10],zd[10];

while()
 a(ad,ab,ax);
 write(ax,10,P2);
 b(ab,bd,by);
 write(by,10,P2);
 read(zd,10,P2);
 d(bd,ad,zd);
```

```
//P2.C
int xc[10],yc[10],cz[10];

while()
 read(xc,10,P1);
 read(yc,10,P1);
 c(xc,yc,cz);
 write(cz,10,P1);
```

FIGURE 5.2. Baseline software synthesis: A) Task Firing Sequence. B) Synthesized software modules. C) Gantt chart. Execution period is $EP = 9$.

`read` tasks that transfer the data between the buffer arrays and the platform FIFO channels (Figure 5.2.B).

Attributes: Tasks and channels are annotated with the following attributes. Computation workload $w(v)$ represents the latency of executing task $v \in V_k$ in processor p_k. Note that the workload assigned to a `write`/`read` task accounts only for transferring tokens from/to the buffer arrays to/from platform FIFO channels, and does not include the wait times due to unavailability of buffer capacity/data in the interconnect fabric.

We also annotate the tasks v and channels e with their memory requirement $m(v)$ and $m(e)$, respectively. That is, $m(v)$ represents the amount of memory required by processor p_k to store the code that implements the data transformation function of a task $v \in V_k$. Similarly, $m(e)$ refers to the amount of memory allocated as an array on processor p_k to store data tokens of channel $e \in E_k$. For example in Figure 5.1, if we assume each data token requires 1 byte of memory, we have $m(bd) = n(bd) \times \text{sizeof}(datatype) = 30 \times 1 = 30$ bytes. Formally,

$$\begin{aligned} w &: V \rightarrow \mathbb{N} \\ m &: E - C \rightarrow \mathbb{N} \end{aligned} \tag{5.10}$$

5.1.2. Memory Requirement. Both tasks and channels take up memory in the synthesized implementation. Specifically, the generated code for processor p_k requires the following amount of memory.

$$\begin{aligned} m(p_k) &= \underbrace{\sum_{v \in V_k} m(v)}_{\textit{task memory}} + \underbrace{\sum_{e \in E_k} m(e)}_{\textit{channel memory}} \\ &= \|m(V_k)\| + \|m(E_k)\| \end{aligned} \tag{5.11}$$

For example in Figure 5.2.B, the channel memory requirement of p_1 is $\|m(E_1)\| = m(ab) + m(ad) + m(ax) + m(bd) + m(by) + m(zd) = 20 + 10 + 10 + 30 + 10 + 10 = 90$ bytes (assuming sizeof($datatype$) $= 1$). Total required memory is

$$\begin{aligned} M &= \sum_{1 \le k \le P} m(p_k) \\ &= \sum_{v \in V} m(v) + \sum_{e \in E-C} m(e) \\ &= \|m(V)\| + \|m(E-C)\| \end{aligned} \tag{5.12}$$

5.1.3. Execution Period. We visualize the steady state execution of a synthesized application with a time-periodic Gantt chart (Figure 5.2.C). In our example, first tasks a, x^1, b, and y^1 execute. As a result the data on ax and by are written to FIFO. At this point, p_2 which was waiting to read this data resumes its operation[2] and task c executes. Next, the data on cz is transferred to p_1, and d which was waiting for this data executes, and this pattern repeats periodically.

Throughput depends on a number of factors such as platform architecture, processors workload and FIFO buffer capacities. We assume that the interconnect fabric provides large buffers, which disentangle the steady state execution of producer and consumer pairs in the steady state [**SGB08**].

[2]We assume latency of transferring a token to FIFO is considerably smaller than that of system calls to `write` or `read`.

Execution period (EP) is the inverse of throughput and quantifies the execution latency of one iteration of the periodic execution. Formally

$$\begin{aligned} EP &= \Delta t(v) = t(v_i) - t(v_{i-1}) \\ Throughput &= 1 \div EP \end{aligned} \tag{5.13}$$

where $t(v_i)$ denotes the time at which task v is fired for the i'th time. We assume i is large enough for the application to exhibit its steady state behavior. In our example $EP = 9$.

Efficient algorithms exist for throughput calculation of dataflow applications [**Thi07, SGB08, WBGB10**]. We use Gantt charts to *visualize* the periodic execution and to facilitate presentation.

5.2. Iteration Overlapping

Given a specific task assignment and scheduling, we propose to synthesize software that overlaps different iterations of the periodic execution in order to potentially improve the throughput at the cost of larger memory requirement.

5.2.1. Motivating Example. As depicted in the example of Figure 5.2, baseline software synthesis may produce code with poor performance, as some processor might block its execution due to data dependencies. For example in processor p_1, execution of task d depends on the availability of data on the cut channel z which has to be received from task c. Therefore p_1 has to stay idle until p_2 completes execution of task c (Figure 5.2).

Overlapping different iterations of the periodic execution provides the opportunity to fill in the idle times with execution of tasks (subject to the given task assignment and schedule) from a different iteration, which can potentially improve the throughput. Figure 5.3 depicts an overlapped version of the example in which, p_1 does not stay idle while waiting for completion of task c. Instead, p_1 proceeds with the execution of tasks a and x^1 from the next iteration of the application execution. In other words, firings of a and x^1 are one iteration ahead of the current loop iteration. Therefore, as visualized in the Gantt chart of Figure 5.3.C, all tasks repeat every 7 time units, which means EP is reduced from 9 to 7.

The improvement in throughput, however, comes at the expense of larger memory requirement. In the example, channel ad requires twice the previously allocated amount of memory. This is because task a is one iteration ahead of the while loop and it executes

(A) **Task Firing Sequence:**

P1: $a_0, x^1_0, \langle b_i, y^1_i, a_{i+1}, x^1_{i+1}, z^1_i, d_i \rangle_{i=0\to\infty}$

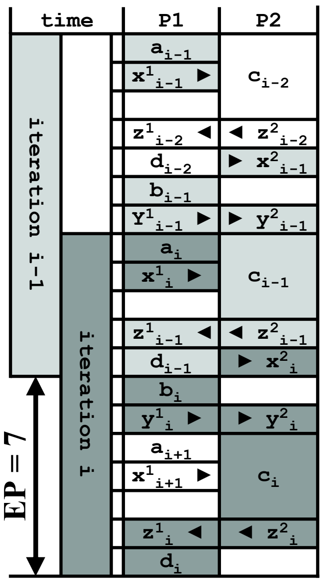
P2: $\langle x^2_i, y^2_i, c_i, z^2_i \rangle_{i=0\to\infty}$

(B) **Code Generation:**

```
//P1.C
int ad[2][10],ab[20],ax[10],
    bd[30],by[10],zd[10];

a(ad[0],ab,ax);              //a0
write(ax,10,P2);             //x0

int i=0;
while()
 b(ab,bd,by);                //bi
 write(by,10,P2);            //yi
 a(ad[(i+1)%2],ab,ax);       //ai+1
 write(ax,10,P2);            //xi+1
 read(zd,10,P2);             //zi
 d(bd,ad[i%2],zd);           //di
 i++;
```

```
//P2.C
int xc[10],yc[10],cz[10];

while()
 read(xc,10,P1);
 read(yc,10,P1);
 c(xc,yc,cz);
 write(cz,10,P1);
```

(C) **Gantt Chart:**



(D)

j	1	2	3	4	5	6
$\sigma_j=\pi_1^{-1}(j)$	b	y^1	a	x^1	z^1	d
$\lambda(\sigma_j)$	0	0	1	1	0	0

j	1	2	3	4
$\sigma_j=\pi_2^{-1}(j)$	x^2	y^2	c	z^2
$\lambda(\sigma_j)$	0	0	0	0

FIGURE 5.3. Overlapping different iterations yields a smaller execution period of $EP = 7$ at the cost of increased memory requirement.

before task d in the loop. Thus, the inter-task memory requirement of processor p_1 is $m(ab)+2m(ad)+m(ax)+m(bd)+m(by)+m(zd) = 20+20+10+30+10+10 = 100$ bytes. Assuming tasks are implemented as function calls, the overhead in instruction memory due to iteration overlapping is negligible.

Note that the increase in throughput might also increase the latency. For example in Figure 5.3, the latency is equal to $t(d_i) - t(a_i) = 12$ which is larger than the original latency of 9. Studying latency is beyond the scope of this work.

5.2.2. Problem Statement. We are interested in the tradeoff between throughput and memory requirement of the synthesized implementations. Specifically, the problem can be stated as:

Given an application task graph and its associated attributes (e.g. workloads w and memory requirements m), task assignment and task scheduling (p and s), what are the bounds on the best possible implementation in terms of throughput or memory requirement? Can we generate a set of design points with smooth throughput-memory tradeoff via iteration overlapping?

5.2.3. Formalism. Iteration overlapping perturbs the periodic task firing sequence $\langle\Sigma_k\rangle_{i=0\to\infty}$ (for all $k \in [1,P]$) to a new task firing sequence $\tilde{\Sigma}_k = \tilde{\Sigma}_k^{in}\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty}$, where $\tilde{\Sigma}_k^{in}$ denotes the initialization and $\tilde{\Sigma}_k^{ss}$ the steady state (periodic) section of $\tilde{\Sigma}_k$:

$$\langle\Sigma_k\rangle_{i=0\to\infty} \xrightarrow{(\lambda,\pi)} \tilde{\Sigma}_k^{in}\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty} \tag{5.14}$$

For example, as shown in Figure 5.3.A, $\tilde{\Sigma}_1$ is equal to

$$a_0, x_0, \langle b_i, y_i^1, a_{i+1}, x_{i+1}^1, z_i^1, d_i\rangle_{i=0\to\infty} \tag{5.15}$$

The above perturbation process is characterized by a tuple (λ,π), where λ and π are the set of $\lambda(v)$ and $\pi(v)$ values for all $v \in V$, respectively. In short, $\lambda(v)$ denotes the lead index of task v, and $\pi(v)$ denotes the position of task v in the periodic firing sequence. The formal definitions are described below.

Steady State Sequence: Based on a given tuple (λ,π), the steady state task firing sequence is defined as

$$\begin{aligned}&\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty} = \\ &\big\langle\sigma_{1,i+\lambda(\sigma_1)}, \sigma_{2,i+\lambda(\sigma_2)}, \ldots, \sigma_{|V_k|,i+\lambda(\sigma_{|V_k|})}\big\rangle_{i=0\to\infty}\end{aligned} \tag{5.16}$$

where σ_j is the j'th task in the periodic firing sequence $\tilde{\Sigma}_k^{ss}$, and $\lambda(\sigma_j)$ denotes the *lead index* of task σ_j which is defined below.

In iteration i of the periodic sequence $\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty}$, $i+\lambda(v)$'th instance of task v is fired, i.e., task v from application iteration $i+\lambda(v)$ is fired. For example in Figure 5.3.A, $\lambda(a) = \lambda(x^1) = 1$ and $\lambda = 0$ for all other tasks. Formally

$$\lambda : V \to \mathbb{N}_0 \tag{5.17}$$

Function π defines a total order among the tasks that are assigned to same processor. Formally

$$\begin{aligned}&\pi : V \to \mathbb{N} \\ &\forall v \in V_k : \pi(v) = \pi_k(v)\end{aligned} \tag{5.18}$$

where

$$\begin{aligned} &\pi_k : V_k \to \{1, 2, \dots, |V_k|\} \\ &\pi_k(v) < \pi_k(u) \iff t(v_{i+\lambda(v)}) < t(u_{i+\lambda(u)}) \end{aligned} \tag{5.19}$$

For example in Figure 5.3.A, $\pi(a) = \pi_1(a) = 3$ because a is the third task in the periodic sequence $\tilde{\Sigma}_1^{ss}$. Since π_k is a one-to-one function, π_k^{-1} is defined and it refers to the task at a certain position in the periodic sequence $\tilde{\Sigma}_k^{ss}$. For example, $\pi_1^{-1}(3) = a$. Formally

$$\begin{aligned} &\pi_k^{-1} : \{1, 2, \dots, |V_k|\} \to V_k \\ &\pi_k^{-1}(j) = v \iff \pi_k(v) = j \end{aligned} \tag{5.20}$$

Therefore, as shown in Figure 5.3.D, σ_j in definition of $\tilde{\Sigma}_k^{ss}$ above, is given by

$$\sigma_j = \pi_k^{-1}(j) \tag{5.21}$$

In baseline software synthesis (no iteration overlapping), the firing sequence is not perturbed which means $\tilde{\Sigma}_k^{ss} = \Sigma_k^{ss}$, hence, $\pi(v) = s(v)$ and $\lambda(v) = 0$ for all tasks.

Initial Sequence: The initial task firing sequence $\tilde{\Sigma}_k^{in}$ is constructed from $\tilde{\Sigma}_k^{ss}$ as the following. $\tilde{\Sigma}_k^{in}$ is a subsequence of the sequence $\langle\tilde{\Sigma}_k^{ss}\rangle_{i=-\infty\to-1}$, i.e., $\tilde{\Sigma}_k^{in} \sqsubset \langle\tilde{\Sigma}_k^{ss}\rangle_{i=-\infty\to-1}$, composed of tasks $v_{i'}$ with $i' \geq 0$:

$$\tilde{\Sigma}_k^{in} = \{v_{i'} \mid v_{i'} \in \langle\tilde{\Sigma}_k^{ss}\rangle_{i=-\infty\to-1}, i' \geq 0\} \tag{5.22}$$

For example in Figure 5.3, $\tilde{\Sigma}_1^{in}$ is equal to

$$\begin{aligned} &\langle b_i, y_i^1, a_{i+1}, x_{i+1}^1, z_i^1, d_i\rangle_{i=-\infty\to-1} \\ &= \dots, b_{-1}, y_{-1}^1, \underline{a_{-1+1}}, \underline{x_{-1+1}^1}, z_{-1}^1, d_{-1} \\ &= a_0, x_0^1 \end{aligned} \tag{5.23}$$

Theorem 5.1. *Let $|\tilde{\Sigma}_k^{in}|$ denote the number of task firings in $\tilde{\Sigma}_k^{in}$. We have*

$$|\tilde{\Sigma}_k^{in}| = \|\lambda(V_k)\| \tag{5.24}$$

Proof. Based on the definition above, each task $v \in V_k$ appears $\lambda(v)$ times in the initial sequence. Hence, the total number is $\sum\limits_{v \in V_k} \lambda(v) = \|\lambda(V_k)\|$. □

5.2.4. Constraints of Valid Overlapping. As described above, the tuple (λ, π) specifies an iteration overlapping instance. However, not all possible values for the tuple yield valid overlappings.

Task Scheduling Constraint: Iteration overlapping should respect the given task assignment and task schedule. In other words, the overall task firing sequence can be perturbed while firing order of the tasks in same application iteration is preserved. Let us distinguish between intra-iteration schedule and inter-iteration schedule.

Intra-iteration schedule is the firing order of tasks from the *same* iteration of the application execution. In other words, it is the given task firing schedule (Figure 5.1.E), and has to be preserved after iteration overlapping. As we see in the Gantt chart in Figure 5.3.C, tasks from the same iteration (same color in the figure) are still in the order of their original steady state schedule, i.e., (a, b, x^1, y^1, z^1, d) for processor p_1 and (x^2, y^2, c, z^2) for processor p_2. The position of a task v in the intra-iteration schedule of processor p_k is given by $s_k(v)$.

Inter-iteration schedule, however, refers to the periodic pattern in the overall firing sequence of tasks from *any* application iteration. In Figure 5.3, the overall task firing sequence in processor p_1 follows a periodic schedule (b, y^1, a, x^1, z^1, d). The position of a task v in the inter-iteration schedule of processor p_k is given by $\pi_k(v)$.

The intra-iteration schedule (a, x^1, b, y^1, z^1, d) is preserved because $\lambda(a) = \lambda(x^1) = 1$ which means tasks a and x^1 are always fired one iteration ahead of the periodic execution (b, y^1, a, x^1, z^1, d), i.e., the firing sequence $\tilde{\Sigma}_1$ is

$$a_0, x_0, b_0, y_0^1, \underline{a_1}, \underline{x_1^1}, z_0^1, d_0, \underline{b_1}, \underline{y_1^1}, a_2, x_2^1, \underline{z_1^1}, \underline{d_1}, \ldots \tag{5.25}$$

Theorem 5.2. *The intra-iteration schedule in processor p_k is preserved if and only if the tuple (λ, π) satisfies the following condition.*

$$\forall v, u \in V_k, s(v) < s(u) : \begin{cases} \lambda(v) = \lambda(u) \text{ and } \pi(v) < \pi(u) \\ or \\ \lambda(v) > \lambda(u) \end{cases} \tag{5.26}$$

PROOF. Intra-iteration schedule in processor p_k is given by s_k. Based on the definition of s_k

$$\begin{aligned} s_k(v) < s_k(u) &\iff t(v_i) < t(u_i) \\ &\iff t(v_{\underbrace{i-\lambda(v)}_{i''}+\lambda(v)}) < t(u_{\underbrace{i-\lambda(u)}_{i'}+\lambda(u)}) \\ &\iff t(v_{i''+\lambda(v)}) < t(u_{i'+\lambda(u)}) \end{aligned} \tag{5.27}$$

As discussed earlier in Section 5.2.3, during iteration $i = l$ of $\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty}$ periodic sequence, $l + \lambda(x)$'th instance of task x, i.e., $x_{l+\lambda(x)}$, is fired. Therefore, $v_{i''+\lambda(v)}$ is fired during iteration i'' and $u_{i'+\lambda(u)}$ during iteration i' of the sequence $\langle\tilde{\Sigma}_k^{ss}\rangle_{i=0\to\infty}$. Hence, the above condition implies that i'' can not be larger than i', because if $i'' > i'$ all task firings in iteration i' are before i'' which is contradicting with above condition. If $i'' = i'$, we have

$$t(v_{i''+\lambda(v)}) < t(u_{i''+\lambda(u)}) \iff \pi(v) < \pi(u) \tag{5.28}$$

and also

$$i - \lambda(v) = i - \lambda(u) \iff \lambda(v) = \lambda(u) \tag{5.29}$$

If $i'' < i'$, we have

$$i - \lambda(v) < i - \lambda(u) \iff \lambda(v) > \lambda(u) \tag{5.30}$$

□

Intuitively, to guarantee v_i fires before u_i, v should fire before u when they both have the same lead index λ, i.e., when v_i and u_i fire in the same iteration. Alternatively, v can have a larger lead index which means v_i fires in an earlier iteration. In short, the ordering should be preserved unless the predecessor has a larger lead index.

COROLLARY 5.3. *$\lambda(v) = 0$ for all $v \in V$ alone characterizes the baseline software synthesis, i.e., no iteration overlapping, because based on the above theorem, no firing order is perturbed and thus $\pi(v) = s(v)$.*

FIFO Scheduling Constraint: Processors communicate by sending and receiving messages through unidirectional FIFO links. Therefore, for an overlapping to be valid, the messages have to be read from each FIFO in the same order written to that FIFO.

For a unidirectional FIFO f from processor $p_{k'}$ to processor $p_{k''}$, let $\tilde{\Sigma}_{k',f}$ denote the firing sequence of `write` tasks in $p_{k'}$ that write to f. $\tilde{\Sigma}_{k',f}$ is a subsequence of $\tilde{\Sigma}_{k'}$, i.e., $\tilde{\Sigma}_{k',f} \sqsubset \tilde{\Sigma}_{k'}$. Similarly, let $\tilde{\Sigma}_{k'',f} \sqsubset \tilde{\Sigma}_{k''}$ denote the firing sequence of tasks in $p_{k''}$ that `read` from f. For the FIFO f from p_1 to p_2 in Figure 5.3, $\tilde{\Sigma}_{1,f} = x_0^1, \langle y_i^1, x_{i+1}^1 \rangle_{i=0\to\infty}$, and $\tilde{\Sigma}_{2,f} = \langle x_i^2, y_i^2 \rangle_{i=0\to\infty}$. An overlapping is valid only if, for all FIFOs, $\tilde{\Sigma}_{k',f}$ is *equivalent* to $\tilde{\Sigma}_{k'',f}$, which is denoted with

$$\tilde{\Sigma}_{k',f} \equiv \tilde{\Sigma}_{k'',f} \tag{5.31}$$

The two write and read sequences are called equivalent when they refer to the same sequence of cut channels $C_f \subset C$ from $p_{k'}$ to $p_{k''}$. In our example, $C_f = \{x, y\}$, and $\tilde{\Sigma}_{1,f} \equiv \tilde{\Sigma}_{2,f}$ because both $x_0^1, \langle y_i^1, x_{i+1}^1 \rangle_{i=0\to\infty}$ and $\langle x_i^2, y_i^2 \rangle_{i=0\to\infty}$ refer to the same cut channel sequence $x, y, x, y, \ldots$

Let c_j's denote the cut channels in C_f according to the order given by the inter-iteration schedule of processor $p_{k''}$. In other words

$$\pi(c_1^{k''}) < \pi(c_2^{k''}) < \ldots < \pi(c_{|C_f|}^{k''}) \tag{5.32}$$

in which, c_j^k denotes the associated write or read task of cut channel c_j in processor p_k. In our example, $c_1 = x$ and $c_2 = y$, hence, $c_1^{k'} = x^1$, $c_2^{k'} = y^1$, $c_1^{k''} = x^2$ and $c_2^{k''} = y^2$.

THEOREM 5.4. *For a unidirectional FIFO f between processors $p_{k'}$ and $p_{k''}$, we have $\tilde{\Sigma}_{k',f} \equiv \tilde{\Sigma}_{k'',f}$ if and only if the tuple (λ, π) satisfies*

$$I)\ \pi(c_{r+1}^{k'}) < \ldots < \pi(c_{|C_f|}^{k'}) < \pi(c_1^{k'}) < \ldots < \pi(c_r^{k'}) \tag{5.33}$$

and

$$II)\ \lambda(c_j^{k'}) = \begin{cases} \lambda(c_j^{k''}) + q & \text{if } r+1 \le j \le |C_f| \\ \lambda(c_j^{k''}) + q + 1 & \text{if } 1 \le j \le r \end{cases} \tag{5.34}$$

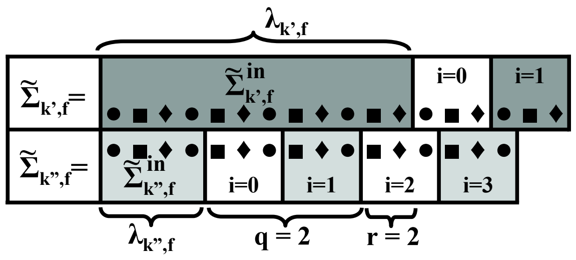


FIGURE 5.4. Example write and read sequences of a FIFO channel f from $p_{k'}$ to $p_{k''}$.

In this theorem, $q \in \mathbb{Z}$ and $r \in [\,0, |C_f|\,)$ are determined based on tuple (λ, π) as

$$\|\lambda_{k',f}\| = \|\lambda_{k'',f}\| + q|C_f| + r \tag{5.35}$$

where $\lambda_{k,f}$ denotes the set of λ values for the write or read tasks, i.e., $\lambda_{k,f} = \{\lambda(c_j^k) | c_j \in C_f\}$, and thus, $\|\lambda_{k,f}\| = \sum_{c_j \in C_f} \lambda(c_j^k)$.

For example in Figure 5.3, $\|\lambda_{1,f}\| = \lambda(x^1) + \lambda(y^1) = 1 + 0 = 1$ and $\|\lambda_{2,f}\| = \lambda(x^2) + \lambda(y^2) = 0 + 0 = 0$. Hence, $r = 1$ and $q = 0$. The FIFO scheduling constraint is satisfied in this example overlapping. Figure 5.4 depicts a larger example.

PROOF. Since the periodic sequences $\tilde{\Sigma}^{ss}_{k',f}$ and $\tilde{\Sigma}^{ss}_{k'',f}$ repeat iteratively, they should be a circular shift of one another in order to have $\tilde{\Sigma}_{k',f} \equiv \tilde{\Sigma}_{k'',f}$ (Figure 5.4). Hence, condition (I) is required, in which r denotes the amount of the circular misplacement. Similar to Theorem 5.1, we have $|\tilde{\Sigma}^{in}_{k,f}| = \|\lambda_{k,f}\|$, and therefore, r is calculated as $\|\lambda_{k',f}\| = \|\lambda_{k'',f}\| + q|C_f| + r$ (Figure 5.4). As a result, the first $|C_f| - r$ write tasks in iteration i of $\tilde{\Sigma}_{k',f}$ match the last $|C_f| - r$ read tasks in iteration $i + q$ of $\tilde{\Sigma}_{k'',f}$, for $i \geq 0$ (The circle task in Figure 5.4). Similarly the other write tasks (The square and diamond tasks) match their corresponding read tasks from one iteration ahead, i.e., $i + q + 1$. Hence, condition (II) is required as well. □

A tuple (λ, π) is valid if and only if both task scheduling and FIFO scheduling constraints are satisfied, i.e., the intra-iteration schedule is preserved for all processors (Theorem 5.2), and the write and read task firing sequences are equivalent for all FIFOs (Theorem 5.4).

5.2.5. Code Generation. A valid tuple (λ, π) determines a valid task firing sequence $\tilde{\Sigma}_k = \tilde{\Sigma}^{in}_k \langle \tilde{\Sigma}^{ss}_k \rangle_{i=0\to\infty}$ for every processor p_k. Given $\tilde{\Sigma}_k$, generating the corresponding parallel code modules is an extension of the baseline code generation, with the enhancement of generating an epilogue. The steady state section of the code (the loop) in processor p_k is generated based on $\tilde{\Sigma}_k^{ss}$, and the initial section based on $\tilde{\Sigma}_k^{in}$ (Figure 5.3.B).

In generating the codes, more than one copy of buffer memory may need to be allocated for a channel $e \in E_k$ if e is alive for more than one iteration, i.e., if i'th instance of the producer task v is fired in an earlier loop iteration than i'th instance of consumer task u. Let $\beta(e)$ denote the number of buffer memories need to be allocated for a channel $e \in E_k$. Formally,

$$\beta : E - C \to \mathbb{N} \tag{5.36}$$

In baseline software synthesis (no iteration overlapping), $\beta(e) = 1$ for all channels. Calculation of β is discussed in Section 5.2.6. In order to preserve the original functionality of the application, i'th instance of task v, i.e., v_i, writes to and u_i reads from copy $i\%\beta(e)$ of the buffer[3]. For example in Figure 5.3.B, channel ad requires two buffers, and hence for any $i \geq 0$, a_i and d_i write to and read from buffer `ad[i%2]`.

5.2.6. Memory Requirement.

THEOREM 5.5. *Given a valid tuple (λ, π), the above code generation procedure maintains original functionality of the application only if*

$$\beta(e) = \begin{cases} \lambda(v) - \lambda(u) + 1 & \textit{if } \pi(v) < \pi(u) \\ \lambda(v) - \lambda(u) & \textit{if } \pi(u) < \pi(v) \end{cases} \tag{5.37}$$

copies of buffer memory is allocated for every channel $e(v, u) \in E_k$.

For example in Figure 5.3, $\beta(ad) = 1 - 0 + 1 = 2$ and $\beta(ab) = 1 - 0 = 1$. Note that, if $\lambda(v) = \lambda(u)$ then $\beta(e) = 1$ because $\pi(v) < \pi(u)$ in such a case.

PROOF. Due to iteration overlapping, between the firing instance v_i and u_i, other instances of v, i.e., $v_{i' \neq i}$, might fire. Since v_i is fired in iteration $i - \lambda(v)$ and u_i in iteration $i - \lambda(u)$, $(i - \lambda(u)) - (i - \lambda(v)) = \lambda(v) - \lambda(u)$ other instances of v are fired between v_i and u_i if $\pi(v) < \pi(u)$, i.e., if v appears before u in the loop. Hence, $\beta(e) = \lambda(v) - \lambda(u) + 1$

[3]% denotes the `mod` operation.

instances of channel $e(v,u)$ are alive at the same time. If $\pi(u) < \pi(v)$, one fewer instance of e is alive because u is fired before v in each iteration of the loop. □

Since we assume that tasks are implemented as function calls, the overhead in instruction memory due to iteration overlapping is negligible. However, the channel memory requirement in processor p_k is updated from $\|m(E_k)\|$ to $\beta(E_k) \bullet m(E_k)$, where the bullet denotes the dot product of two vectors. In other words, the memory requirement is updated from $\sum_{e \in E_k} m(e)$ to $\sum_{e \in E_k} \beta(e)m(e)$.

Corollary 5.6. *In face of iteration overlapping, Equation 5.12 for memory requirement is updated to*

$$M = \|m(V)\| + \beta(E - C) \bullet M(E - C) \tag{5.38}$$

5.3. Throughput-Memory Tradeoff

As mentioned in the problem statement, we are interested in a set of design points with smooth throughput-memory tradeoff.

5.3.1. Tradeoff Bounds. Given an application task graph and its associated attributes (e.g. workloads w and memory requirements m), task assignment and task scheduling (p and s), we proceed to highlight the quality bounds of the design points, i.e., the smallest execution period and the smallest memory requirement that can be achieved via iteration overlapping.

Theorem 5.7. *For a given task assignment, the execution period cannot be smaller than*

$$EP_{\min} = \max_{1 \le k \le P} w(p_k) \tag{5.39}$$

where $w(p_k)$ is the workload of processor p_k which is the sum of its tasks' workloads, i.e., $w(p_k) = \|w(V_k)\| = \sum_{v \in V_k} w(v)$.

In our example, $w(p_1) = w(a)+w(b)+w(x^1)+w(y^1)+w(z^1)+w(d) = 1+1+1+1+1+1 = 6$ and $w(p_2) = w(x^2) + w(y^2) + w(c) + w(z^2) = 1 + 1 + 3 + 1 = 6$. Hence, $EP_{min} = 6$.

Proof. In steady state, each task is fired once in the periodic firing, and therefore, no processor can repeat its steady state execution faster than its assigned workload. That is, it takes at least $w(p_k)$ for processor p_k to execute its task set, even without any blocks due to data dependencies. Therefore, the maximum processor workload is a lower bound on the periodic execution latency. □

THEOREM 5.8. *Given a task assignment and task schedule, there exist a valid iteration overlap* (λ, π) *such that* $EP = EP_{\min}$.

PROOF. Let us combine the given task schedules (intra-iteration schedules) $s_k : V_k \rightarrow \{1, 2, \ldots, |V_k|\}$ into a global ordering of all tasks $\delta : V \rightarrow \{1, 2, \ldots, |V|\}$, such that

$$\begin{aligned} &\forall v, u \in V_k \ , \ s_k(v) < s_k(u) \ : \ \delta(v) < \delta(u) \\ &\forall e(v, u) \in C \ : \ \delta(v) < \delta(u) \end{aligned} \tag{5.40}$$

In our example, $a, x^1, b, y^1, x^2, y^2, c, z^2, z^1, d$ would be the ordering given by δ, in which $\delta(a) = 1, \ldots, \delta(d) = 10$. Next, we assign

$$\begin{aligned} &\pi(v) = s(v) \\ &\lambda(v) = \begin{cases} \lambda(u) + 1 & \text{if } e(v, u) \in C \\ |V| - \delta(v) & \text{otherwise} \end{cases} \end{aligned} \tag{5.41}$$

We claim the above assignment for tuple (λ, π) satisfies both task scheduling and FIFO scheduling constraints, and also, yields the execution period $EP_{\min}$. Let us consider the following lemma as part of the proof.

LEMMA 5.9. *Let* wr *denote a write task which is connected to a read task* rd *through a channel* $e \in C$*, i.e.,* $e(wr, rd) \in C$*. We have* $\lambda(wr) \leq |V| - \delta(wr)$.

The lemma holds because

$$\begin{aligned} e(wr, rd) \in C &\Rightarrow \delta(wr) < \delta(rd) \\ &\Rightarrow \delta(wr) \leq \delta(rd) - 1 \\ &\Rightarrow |V| - \delta(wr) \geq |V| - \delta(rd) + 1 \\ &\Rightarrow |V| - \delta(wr) \geq \lambda(rd) + 1 \\ &\Rightarrow |V| - \delta(wr) \geq \lambda(wr) \end{aligned} \tag{5.42}$$

Now, we resume the proof of Theorem 5.8. The above assignment for tuple (λ, π) yields the execution period $EP_{\min}$ because it fully disentangles the execution of tasks from the same periodic loop iteration. On all processors, for any channel $e(v,u) \in E_k$, i'th instance of task v, i.e., v_i, fires in an earlier iteration than u_i, which means there would be no wait times due to unavailability of data. Formally, we have

$$\begin{aligned} \forall e(v,u) \in C &\Rightarrow \lambda(v) = \lambda(u) + 1 \\ &\Rightarrow \lambda(v) > \lambda(u) \end{aligned} \tag{5.43}$$

and we also have

$$\begin{aligned} e(v,u) \in E_k &\Rightarrow s_k(v) < s_k(u) \\ &\Rightarrow \delta(v) < \delta(u) \\ &\Rightarrow \underbrace{|V| - \delta(v)}_{=\lambda(v)} > \underbrace{|V| - \delta(u)}_{\geq\lambda(u)} \\ &\Rightarrow \lambda(v) > \lambda(u) \end{aligned} \tag{5.44}$$

Since v is not a `write` task, we have $|V| - \delta(v) = \lambda(v)$, and since u might or might not be a `write` task, based on above lemma, we have $|V| - \delta(u) \geq \lambda(u)$. Therefore, for any channel, the producer task has a larger lead index (I).

This assignment satisfies the task scheduling constraint (Theorem 5.2) because as we explained above for any $s_k(v) < s_k(u)$ we have $\lambda(v) > \lambda(u)$ (II). It also satisfies the FIFO scheduling constraint (Theorem 5.4). Since $\pi(v) = s(v)$, the given periodic task schedule is not perturbed, and therefore, $r = 0$ in Theorem 5.4. Since $\lambda(v) = \lambda(u) + 1$, for all cut channels $e(v,u)$, we have

$$\|\Lambda_{k'',f}\| = \sum_{c_j \in C_f} \lambda(c_j^{k''}) = \sum_{c_j \in C_f} \lambda(c_j^{k'}) + 1 = \|\Lambda_{k',f}\| + |C_f| \tag{5.45}$$

and thus $q = 1$ (III). □

Corollary 5.10. *Iteration overlapping enables implementing the application with the optimal execution period of $EP = EP_{\min}$, which gives the highest throughput.*

Theorem 5.11. *For a given scheduling and assignment of tasks, zero iteration overlapping, i.e., $\lambda(v) = 0$ for all $v \in V$, results in the minimum memory requirement which is* *given by*

$$M_{min} = \|M(v)\| + \|M(E-C)\| \tag{5.46}$$

Proof. Based on Theorem 5.5, $\beta(e) = 1$ for all channels if $\lambda(v) = 0$ for all tasks. Since we always have $\beta \geq 1$, this is the minimum possible value for β, and in Equation 5.38, the term $\beta(E-C) \bullet M(E-C)$ is minimized to $\|M(E-C)\|$. □

5.3.2. Generation of Competitive Design Points. In this section we present an algorithm that generates a set of design points obtained using a corresponding set of valid tuples (λ, π) with smooth throughput-memory tradeoff.

Gantt Chart Construction: First we explain a heuristic method for iteration overlapping which generates a valid tuple (λ, π) for a given $EP \geq EP_{\min}$, without excessive overlapping of iterations[4]. The algorithm is explained with reference to the example in Figure 5.5 where $EP = EP_{\min} = 6$.

Let us view the time-periodic Gantt chart as the starting point. An iteration overlapping can be thought of as a placement of tasks in time. Every task $v \in V$ can be viewed as an object of height $w(v)$ which has to be placed in $p(v)$'s timeline, subject to the constraints. Once such an execution timeline is available, the corresponding tuple (λ, π) is readily available.

Given task assignment and schedule, and $EP \geq EP_{\min}$, our algorithm constructs a Gantt chart as the following. We visit the tasks in the reverse of the δ ordering (Theorem 5.8), and place them into their corresponding processor timeline (Figure 5.5.A). Specifically, for every task $v \in V_k$, an object v_i is placed as far down as possible in the execution timeline p_k, as long as it does not violate the following conditions

$$\begin{aligned} &\forall v, u \in V_k \ , \ s_k(v) < s_k(u) \ : \ t(v_i) < t(u_i) \\ &\forall e(v,u) \in C \ : \ t(v_i) \leq t(u_i) \end{aligned} \tag{5.47}$$

where $t(v_i) < t(u_i)$ means v_i is placed higher than, i.e., before, u_i. For example in Figure 5.5.A, z_i^2 in column p_2 can not be placed any lower than depicted in the figure because of the placement of z_i^1.

[4]The constructive proof of Theorem 5.8 will result in an implementation with unreasonably large memory requirement.

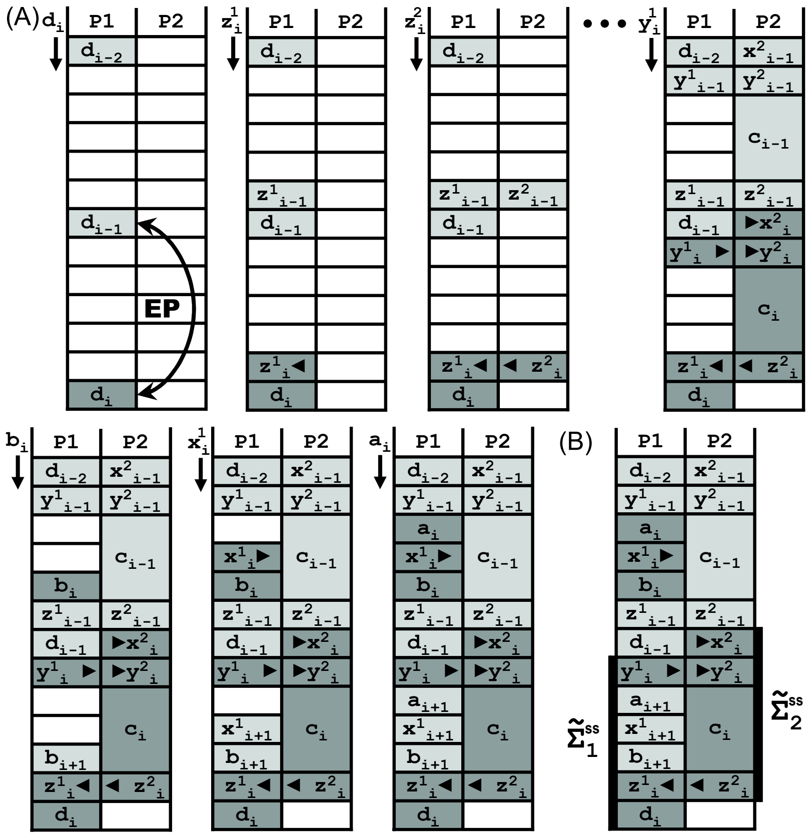
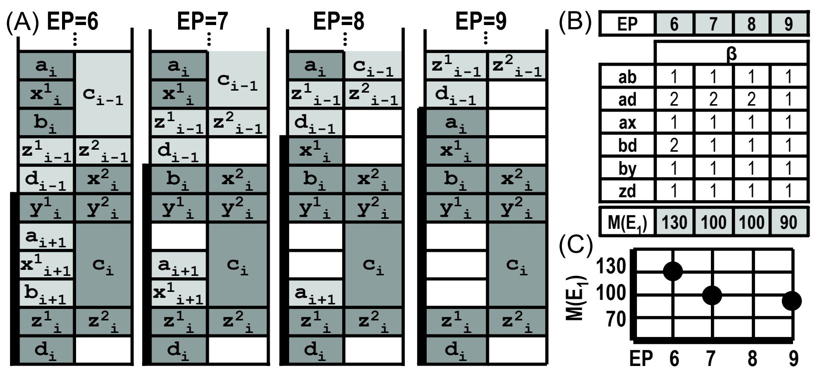


FIGURE 5.5. A) Construction of a time-periodic Gantt chart with $EP = EP_{\min} = 6$. The ordering δ is $a, x^1, b, y^1, x^2, y^2, c, z^2, z^1, d$. B) The chart determines firing sequences $\tilde{\Sigma}_k^{ss}$ which is marked by black bars, and $\tilde{\Sigma}_k^{ss}$ gives us the tuple (λ, π).

When an object v_i is placed at a position $t(v_i)$, all positions with integral multiples of EP time distance from v_i are marked as unavailable (occupied with other instances of v), which ensures the constructed Gantt chart has the desired steady state execution period.

In addition, visiting of the tasks in the reverse of the δ order and placing them without violating the above condition ensure that the intra-iteration schedule is preserved, i.e., the task scheduling constraint (Theorem 5.2) is satisfied.

Once the Gantt chart is constructed, calculation of the tuple (λ, π) is straight-forward. The chart determines firing sequences $\tilde{\Sigma}_k^{ss}$ for all $1 \leq k \leq P$. As defined in Section 5.2.3, $\tilde{\Sigma}_k^{ss}$ gives us the $\lambda(v)$ and $\pi(v)$ of all tasks $v \in V_k$. In Figure 5.5.B, $\tilde{\Sigma}_1^{ss} = y_i^1, a_{i+1}, x_{i+1}^1, b_{i+1}, z_i^1, d_i$, and hence, $\lambda(a) = 1$ because a_{i+1} is fired in iteration i of $\tilde{\Sigma}_1$, and $\pi(a) = 2$ because a is the second task in $\tilde{\Sigma}_1^{ss}$.



EP	6	7	8	9
	β			
ab	1	1	1	1
ad	2	2	2	1
ax	1	1	1	1
bd	2	1	1	1
by	1	1	1	1
zd	1	1	1	1
$M(E_1)$	130	100	100	90

FIGURE 5.6. A) The Gantt charts with different execution periods EP. The black bar marks $\tilde{\Sigma}_1^{ss}$. B) Channel memory requirement of p_1. C) The generated set of competitive design points.

The example in Figure 5.5 satisfies the FIFO scheduling constraint (Theorem 5.4), but the above procedure might result in a tuple (λ, π) which does not satisfy the constraint. This is rare in practice because graph partitioning algorithms employed for task assignment often split a given task graph into a set of *connected* sub-graphs. If the constraint is not satisfied for a FIFO f from processor $p_{k'}$ to $p_{k''}$, we adjust π and λ values of the corresponding read tasks of f, i.e., tasks $c_j^{k''}$ for all $c_j \in C_f$. Specifically, we look at the π ordering of the write tasks $c_j^{k'}$ and reorder the read tasks to match this order. Next, we increase the λ values of the read tasks as required to ensure both constraints are satisfied. Note that this correction procedure might change the execution period. Therefore, for a given starting EP, the above Gantt chart construction heuristic provides a tuple (λ, π) and its corresponding execution period EP' which might be different from EP in rare cases.

Design Points: The required memory (M) for a given iteration overlapping (λ, π) can be calculated using Equation 5.38. Hence, the above Gantt chart construction procedure is repeated for different values of EP, in order to generate a set of design points (EP', M) with competitive throughput memory characteristics (Figure 5.6.A). One can start from $EP_{\min}$ and increment EP until the memory requirement M reaches M_{min}.

However, not all EP values need to be tried out. We increase EP in discrete steps that might be larger than 1. Based on Equation 5.38, the memory requirement M is reduced when at least one of the $\lambda(v)$ or $\pi(v)$ values is changed. At each step EP is increased by the minimum amount that is likely to perturb the iteration overlap generated by the above

Gantt chart construction heuristic. In our example, all EP values are tried out. Figure 5.6.B shows the calculation of channel memory requirement of processor p_1 as an example.

Finally, we filter the set of design points (EP', M) and only keep the ones with smooth throughput-memory tradeoff, i.e., the design points whose memory requirements are reduced with increase in execution period (Figure 5.6.C).

5.4. Empirical Evaluation

To demonstrate the merits of our idea, we experimented with the benchmark applications listed in Figure 5.7. For every application, we consider 4, 8 and 16 core target platforms. Given the application and number of cores, we first perform task assignment and task scheduling, and then, apply our iteration overlapping method in synthesizing the parallel software codes as described in Section 5.3.2. In other words, we synthesize different sets of software codes for different execution period values.

Throughput of the synthesized codes are measured on the following FPGA-prototyped system. Each processor is an Altera NiosII/f core with 8KB instruction cache and 32KB data cache. The communication network is a mesh which connects the neighbor processors with FIFO channels of depth 1024. The processors use a shared DDR2-800 memory, but they only access their own region in this memory, i.e., they communicate only through the FIFO channels. The compiler is `gcc` in Altera NiosII IDE with optimization flag `-O2`. The memory requirements are measured by compiling the applications with `gcc` in Altera Nios IDE [**Nio**].

Experiment Results: The measurement results are shown in Figure 5.8. The black curve is the tradeoff points between execution period (X axis) and memory requirement (Y1 axis), for the benchmark applications. Both X and Y1 axis are normalized against EP_{min} and M_{min}, respectively. The gray curve (Y2 axis) is the average value of λ which

Benchmark Name	$\|V\|$	$\|E\|$
Advanced Encryption Standard (AES)	93	102
Block Matrix Multiplication	98	121
Fast Fourier Transform (FFT)	80	110
Parallel Merge Sort	96	126

FIGURE 5.7. Benchmark applications. $|V|$ and $|E|$ denote the number of task graph vertices and edges.

characterizes the amount of overlapping at every design point. When $avg(\lambda) = 0$, we have $\lambda(v) = 0$ for all tasks, i.e, no overlapping.

We observe that memory requirements are decreased with smaller throughput values, i.e., larger execution periods. Our experiments confirm that the set of tradeoff points do not show excessive increase in memory requirement when throughput is increased. For example, the AES application on a 64-core target (Figure 5.8.A, right) shows less than 15% increase in memory requirment when throughput is increased by a factor of 11, i.e., the execution period without iteration overlapping is 11 times the minimum execution period EP_{min}. On average, throughput is increased by a factor of 3.4× for only a 30% increase in memory requirement.

As we see in the figure, our heuristic algorithm finds a set of design points with far less memory requirement comparing to the excessive overlapping of Theorem 5.8. We also observe that the number of tradeoff points decreases with increasing number of cores. This is because fewer tasks are assigned to each processor, hence there is less room for our algorithm to explore the tradeoff points.

5.5. Related Work and Chapter Summary

The problem of iteration overlapping is closely related to that of software pipelining, which is utilized in conventional compilers [**Rau94, RGSL96**] and high-level synthesis frameworks [**Mic94, BWH$^+$03**]. However, traditional software pipelining is designed for only one processor and does not consider the effect of inter-processor communications in overlapping of the iterations.

Gordon et. al. [**GTA06**] proposed an iteration overlapping method, a.k.a, coarse-grain stream software pipelining, in which execution of dependent filters (tasks) are completely decoupled by being assigned to different iterations of the stream program (different λ value for each task), even if they reside in the same processor. Kudlur and Mahlke [**KM08**] proposed a better algorithm for decoupling of task executions, in which the tasks are scheduled in a flat single appearance topological order, and assigned to stages larger than or equal to their producer stage (smaller than or equal to the producer λ value).

The iteration overlapping method discussed in this chapter considers the constraints which exist for targets with FIFO-based point-to-point interprocessor communication, rather

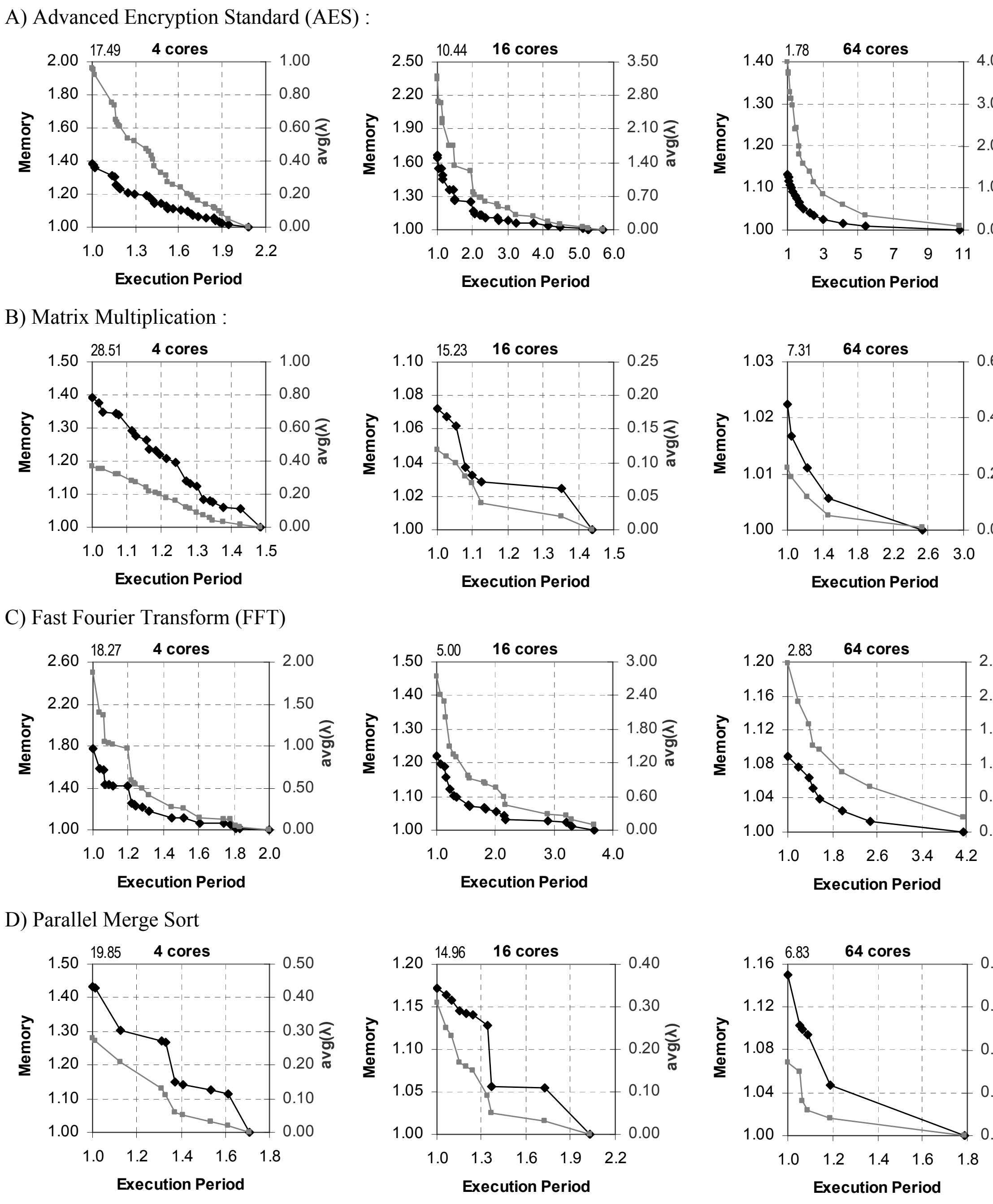


Figure 5.8. Black curve: tradeoff points between execution period (X axis) and memory requirement (Y1 axis), for the benchmark applications. Both X and Y1 axis are normalized against EP_{min} and M_{min}, respectively. Gray curve: the average value of λ (Y2 axis). The three columns show the results for 4, 8 and 16-core targets. The number written on top of each chart is the memory requirement for the excessively overlapped instance as described in Theorem 5.8.

than router or DMA based communications. In addition, our method supports overlapping of iterations for an arbitrary flat single appearance schedule of tasks, rather than topological ordering. Future directions include overlapping of the iterations for any arbitrary task schedule, rather than only single appearance.

CHAPTER 6

Throughput Scaling via Malleable Specification

In principle, specifying the application as a set of tasks and their dependencies is meant to only model the functional aspects of an application, which should enable seamless portability to new platforms by fresh platform-driven allocation and scheduling of tasks and their executions. However, such specifications are rather *rigid* in that some non-behavioral aspects of the application are *implicitly* hard coded into the model at design time. Consequently, the task assignment and task scheduling processes are likely to generate poor implementations when one tries either to port the application to different platforms, or to explore implementation design space on a range of platform choices [**SVCDBS04**]. The limitations of conventional dataflow-based models with portability, scalability and subsequently the ability to explore implementation tradeoffs (e.g., with respect to number of cores) have become especially critical with availability of platforms with a large number of processor cores, which can dedicate a wide range of resources to an application [**TCM+08, BEA+08**].

As an example, consider the merge sort dataflow network, which is composed of actors for splitting the data into segments, sorting of data segments using a given algorithm (e.g., quicksort), and merging of the sorted segments into a unified output stream. A specific instance of the sort network would have rigid structural properties, such as number of sort actors or fanin degree of merge actors. The choice of structure, although implicitly hard coded into the specification, is orthogonal to application's end-to-end functional behavior. It is intuitively clear that the optimal network structure would depend on the target platform, and automatic software synthesis from a rigid specification is bound to generate poor implementations over a range of platforms.

Our driving observation is that the scalability limitation of software synthesis from rigid dataflow models could be addressed if the specifications were sufficiently malleable at compile time, while maintaining functional consistency. We present an example manifestation of the idea, dubbed FORMLESS, which extends the classic notion of dataflow by abstracting away some of the unnecessary structural rigidity in the model. In particular, malleable aspects of the dataflow structure are modeled using a set of parameters, referred to as the *forming vector*. Assignment of values to forming set parameters instantiates a particular structure of the model, while all such assignments lead to the same end-to-end functional behavior. A simple example of a forming set parameter is the fanin degree of merge actors in the sort example.

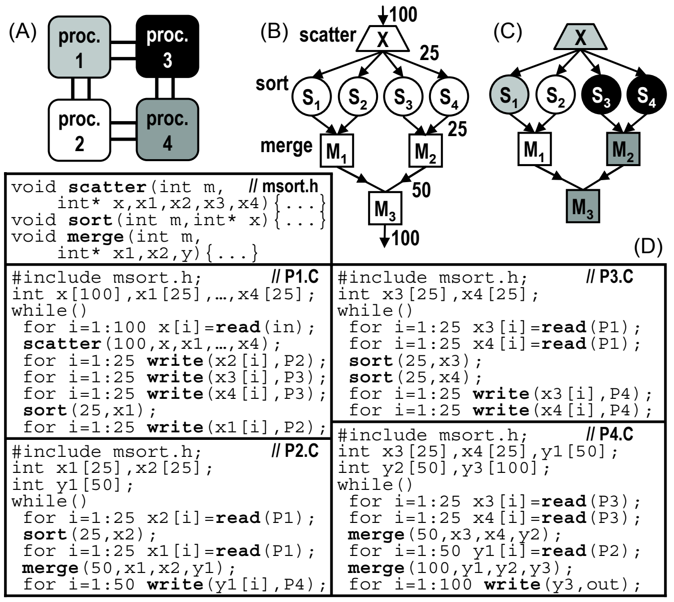
Our approach opens the door to design space exploration methodologies that can *hammer out* a FORMLESS specification to form an optimized version of the model for the target platform. The "formed" model can be subsequently passed onto conventional allocation and scheduling processes to generate a quality parallel implementation. We also present such a design space exploration scheme that determines the forming set using platform-driven profiles of application tasks. Experimental results demonstrate that FORMLESS yields substantially improved portability and scalability over conventional dataflow modeling [**HG11b, HFC$^+$11**].

6.1. Motivating Example

Figure 6.1.A shows an example abstract target platform with four processors. Figure 6.1.B illustrates the SDF graph for an example streaming sort application, which sorts 100 data tokens per invocation. The `scatter` task reads 100 tokens from the input stream, and divides them into segments of 25 tokens that are passed onto the four `sort` tasks. After the four segments are sorted by the `sort` tasks, two `merge` tasks combine the four segments into two larger sorted data segments of size 50. Finally, another `merge` task combines the two segments and generates the sorted output stream. A sample task assignment is shown in Figure 6.1.C, and the generated parallel software modules are shown in Figure 6.1.D.

To motivate the underlying idea of FORMLESS, we investigate the scaling of throughput when platforms with different number of processors are targeted. Let us assume that the `sort` task implements the quicksort algorithm.

An immediate observation is that the example task graph cannot readily utilize many (more than 8 in the case of depicted task graph) processors due to the limited concurrency in the specification. At the other extreme, the throughput of the synthesized software is



```
void scatter(int m,                // msort.h
    int* x,x1,x2,x3,x4){...}
void sort(int m,int* x){...}
void merge(int m,
    int* x1,x2,y){...}
```

```
#include msort.h;                  // P1.C
int x[100],x1[25],…,x4[25];
while()
 for i=1:100 x[i]=read(in);
 scatter(100,x,x1,…,x4);
 for i=1:25 write(x2[i],P2);
 for i=1:25 write(x3[i],P3);
 for i=1:25 write(x4[i],P3);
 sort(25,x1);
 for i=1:25 write(x1[i],P2);
```

```
#include msort.h;                  // P2.C
int x1[25],x2[25];
int y1[50];
while()
 for i=1:25 x2[i]=read(P1);
 sort(25,x2);
 for i=1:25 x1[i]=read(P1);
 merge(50,x1,x2,y1);
 for i=1:50 write(y1[i],P4);
```

```
#include msort.h;                  // P3.C
int x3[25],x4[25];
while()
 for i=1:25 x3[i]=read(P1);
 for i=1:25 x4[i]=read(P1);
 sort(25,x3);
 sort(25,x4);
 for i=1:25 write(x3[i],P4);
 for i=1:25 write(x4[i],P4);
```

```
#include msort.h;                  // P4.C
int x3[25],x4[25],y1[50];
int y2[50],y3[100];
while()
 for i=1:25 x3[i]=read(P3);
 for i=1:25 x4[i]=read(P3);
 merge(50,x3,x4,y2);
 for i=1:50 y1[i]=read(P2);
 merge(100,y1,y2,y3);
 for i=1:100 write(y3,out);
```

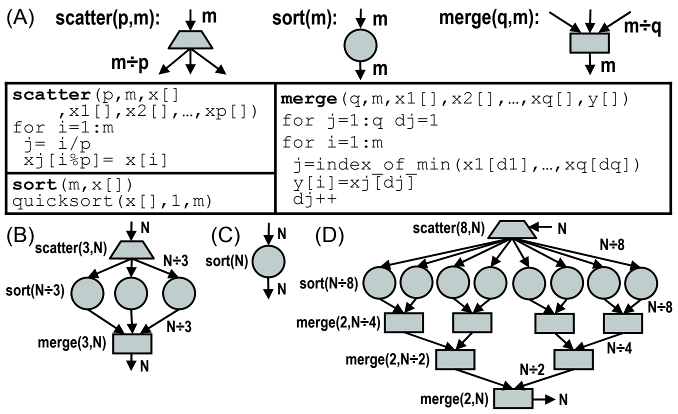
FIGURE 6.1. A) Example platform. B) Sort application modeled as a SDF. C) Tasks are assigned to processors (color coded). D) Synthesized software modules. Outputs of tasks S_i and M_i are implemented with arrays `xi` and `yi`, respectively.

going to be poor when one processor is targeted, compared to eliminating the scatter and merge tasks and running a single sort task (i.e., the quicksort algorithm) on the entire input stream [1]. This is partly because the overhead of inter-task communication is only justified if sufficient amount of parallelism exists in the platform. Intuitively, increasing concurrency in the task graph specification facilitates utilization of more parallel resources and potentially increases the potential for improving performance via load balancing between processors, however, it comes at the cost of degraded performance when platforms with fewer processors are targeted.

Having made this observation, our idea is to specify the tasks and their composition using a number of parameters. Adjustment of parameters enables "massaging" the structure of the task graph to fit the target architecture, while all candidate task graphs deliver the same end to end functionality.

Figure 6.2 sketches the idea for the example sort application in which fanout degree of the scatter task and fanin degree of the merge task are parametrically specified. The number of tasks, their type and composition, as well as their data production rates are

[1]The discussion does not pertain to sorting of large databases which does not entirely fit in the memory.



```
scatter(p,m,x[]
    ,x1[],x2[],…,xp[])
for i=1:m
 j= i/p
 xj[i%p]= x[i]
```

```
sort(m,x[])
quicksort(x[],1,m)
```

```
merge(q,m,x1[],x2[],…,xq[],y[])
for j=1:q dj=1
for i=1:m
 j=index_of_min(x1[d1],…,xq[dq])
 y[i]=xj[dj]
 dj++
```

Figure 6.2. FORMLESS specification of the sort example: A) Actor specifications. B-D) Example instantiations.

immediate functions of the two scatter-fanout and merge-fanin parameters. Three example instances of the FORMLESS graphs are shown in Figure 6.2.

6.2. FORMLESS: A Malleable Dataflow Model

6.2.1. Formalism. We make the key observation that SDF specifications are structurally rigid. Such task graphs do not fully live up to the intended promise of separating functional aspects of the application from implementation platform, and thus, fail to deliver efficient portability and scalability with respect to number of processors in the platform. To address the portability and scalability limitations, not only application specification has to be sufficiently separated from implementation platform, but it also has to admit platform-driven transformations and optimizations.

We propose raising the level of abstraction in specifications to eliminate the rigid structure of the task graph, while preserving its functional behavior. Our approach is to require application designers to specify the tasks and the structure of the task graph using a number of parameters, referred to as the *forming vector*. Specifically, a forming vector Φ is defines as

$$\Phi = (\phi_1, \phi_2, \ldots, \phi_{|\Phi|}) \tag{6.1}$$

where ϕ_j is a *forming parameter* with domain δ_j, i.e., ϕ_j is a member of a set δ_j. Hence, domain of the forming vector Φ is equal to

$$\Delta = \delta_1 \times \delta_2 \times \ldots \times \delta_{|\Phi|} \tag{6.2}$$

We extend the definition of a task α such that input ports, output ports and data transformation function of α are all specified as functions of the underlying parameters in Φ. In other words, task α is defined as the tuple

$$\forall \Phi \in \Delta_\alpha : \ \alpha(\Phi) = \big(In_\alpha(\Phi), Out_\alpha(\Phi), F_\alpha(\Phi)\big) \tag{6.3}$$

For example, the merge task in Figure 6.2.A is defined based on the forming vector $\Phi = \{q, m\}$. The function $In_{merge}(q, m)$ specifies q input ports of rate $\frac{m}{q}$, and function $Out_{merge}(q, m)$ specifies one output port of rate m. The data transformation function $F_{merge}(q, m)$ specifies a mergesort algorithm which combines q sorted input arrays of size $\frac{m}{q}$ into a single sorted output array of size m. In this example, $\Delta_{merge} = \{(q, m) \mid m \geq 2, q \geq 2, m \mod q = 0\}$.

We also extend the definition of task graph $G(V, E)$ such that tasks (V) and channels (E) are specified as functions of the underlying parameters in Φ. Formally, task graph G is defined as the tuple

$$\forall \Phi \in \Delta_G : \ G(\Phi) = \big(V_G(\Phi), E_G(\Phi)\big) \tag{6.4}$$

$V_G(\Phi)$ is a function which specifies the set of tasks in G based on a given forming vector Φ, and is formally defined as

$$V_G(\Phi) = \big\{\alpha_1(\Phi_1), \alpha_2(\Phi_2), \ldots, \alpha_{|V|}(\Phi_{|V|})\big\} \tag{6.5}$$

where $\alpha_i(\Phi_i)$ is an instance of task α_i which is formed based on forming vector Φ_i, and both α_i and Φ_i are determined based on the given Φ. For instance, the task graph in Figure 6.2.B is specified based on forming vector $\Phi = \{p, q, m\} = \{3, 3, N\}$, and function V_G specifies the set of tasks as

$$\begin{aligned} V_G(3,3,N) = \big\{&\text{scatter}(3,N), \text{sort}(\frac{N}{3}), \\ \text{sort}(\frac{N}{3}), \text{sort}&(\frac{N}{3}), \text{merge}(3,N)\big\} \end{aligned} \tag{6.6}$$

in which, for example, task merge$(3, N)$ is an instance of merge$(q, m) = \big(In_{\text{merge}}(q, m)$, $Out_{\text{merge}}(q, m)$, $F_{\text{merge}}(q, m)\big)$, where $\{q, m\} = \{3, N\}$.

Similarly, $E_G(\Phi)$ is a function which specifies the set of channels in G based on the forming vector Φ, and is formally defined as

$$E_G(\Phi) = \big\{(prd, cns) \mid prd \in Out_{\alpha_i}(\Phi_i), cns \in In_{\alpha_j}(\Phi_j)\big\} \tag{6.7}$$

where (prd, cns) denotes a channel from an output port prd of some task α_i to an input port cns of some task α_j.

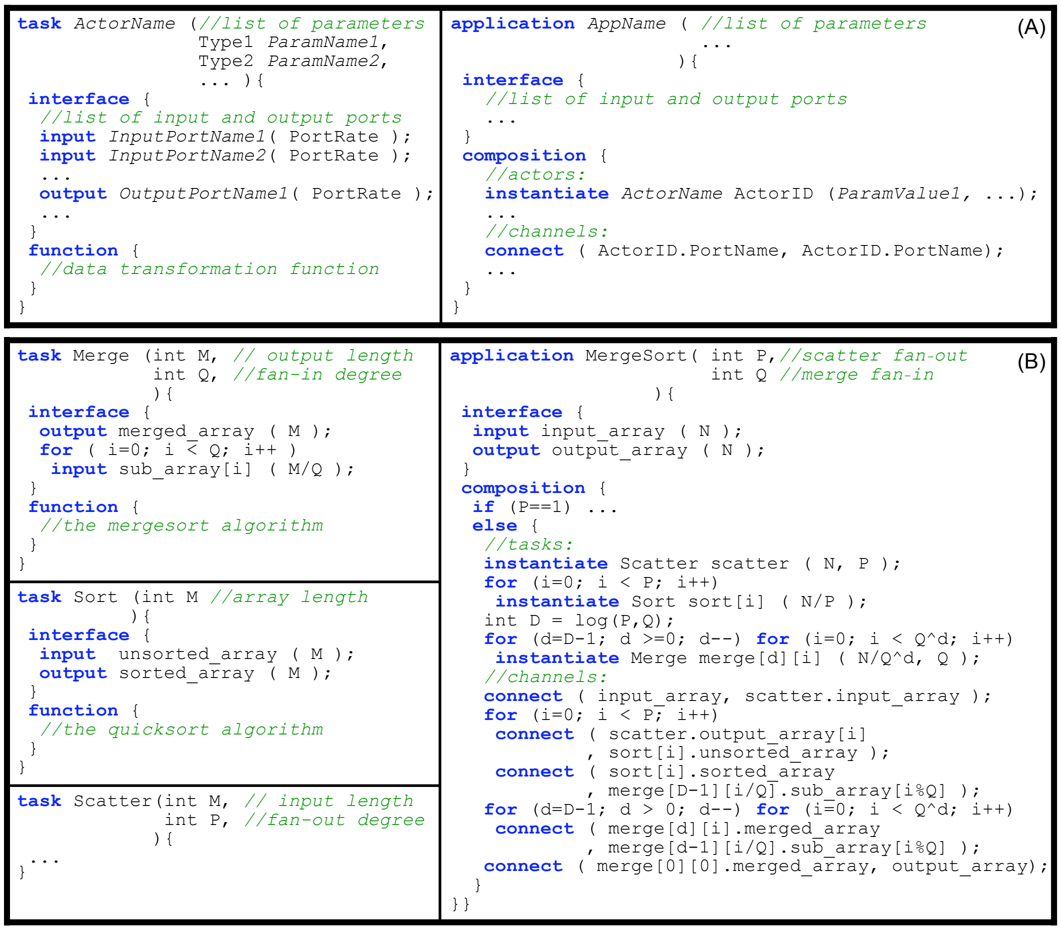
We would like to stress that our primary objective in this work is to demonstrate the merit of the idea and scalability of malleable specifications. In our scheme, it is the programmer's duty to define the ports, task computations and graph composition based on the parameters. Furthermore, he has to ensure that every assignment of values from the specified domain Δ_G to the forming vector Φ results in the same functional behavior. This tends to be straight forward since tasks perform the same high-level function under different parameters (e.g. scattering, sorting or merging in the example of Figure 6.2).

6.2.2. Higher-Order Language. Development of a formal higher-order programming language involves many considerations that are beyond the scope of this work [**CGHP04, Tah03, Cat06**]. However, in this section we present an example realization of the general idea that we have developed.

Figure 6.3.A presents the prototype for specifying task and application task graph based on a set of parameters. The task specification starts with a list of forming parameters and their type. The `interface` section specifies the set of input and output ports of the task, and the `function` section specifies its data transformation function, all based on the given parameters.

Similarly, application specification also starts with a list of forming parameters. The `interface` section is the same as task interface. In a `composition` section, the tasks are instantiated by assigning the corresponding parameters using the `instantiate` construct. The channels are instantiated using the `connect` construct which connects ports of two tasks.

Figure 6.3.B shows the code for our previously mentioned sort application. For example, the merge task is specified with two parameters m and q. As we see the number and rate of input ports in this task is defined using a `for` loop. In general we allow a rich set of


(A)

```
task ActorName (//list of parameters
                Type1 ParamName1,
                Type2 ParamName2,
                ... ){
 interface {
  //list of input and output ports
  input InputPortName1( PortRate );
  input InputPortName2( PortRate );
  ...
  output OutputPortName1( PortRate );
  ...
 }
 function {
  //data transformation function
 }
}
```

```
application AppName ( //list of parameters
                        ...
                      ){
 interface {
   //list of input and output ports
   ...
 }
 composition {
   //actors:
   instantiate ActorName ActorID (ParamValue1, ...);
   ...
   //channels:
   connect ( ActorID.PortName, ActorID.PortName);
   ...
 }
}
```

(B)

```
task Merge (int M, // output length
            int Q, //fan-in degree
            ){
 interface {
  output merged_array ( M );
  for ( i=0; i < Q; i++ )
   input sub_array[i] ( M/Q );
 }
 function {
  //the mergesort algorithm
 }
}
```

```
task Sort (int M //array length
          ){
 interface {
  input  unsorted_array ( M );
  output sorted_array ( M );
 }
 function {
  //the quicksort algorithm
 }
}
```

```
task Scatter(int M, // input length
             int P, //fan-out degree
            ){
 ...
}
```

```
application MergeSort( int P,//scatter fan-out
                       int Q //merge fan-in
                     ){
 interface {
  input input_array ( N );
  output output_array ( N );
 }
 composition {
  if (P==1) ...
  else {
   //tasks:
   instantiate Scatter scatter ( N, P );
   for (i=0; i < P; i++)
    instantiate Sort sort[i] ( N/P );
   int D = log(P,Q);
   for (d=D-1; d >=0; d--) for (i=0; i < Q^d; i++)
    instantiate Merge merge[d][i] ( N/Q^d, Q );
   //channels:
   connect ( input_array, scatter.input_array );
   for (i=0; i < P; i++)
    connect ( scatter.output_array[i]
            , sort[i].unsorted_array );
    connect ( sort[i].sorted_array
            , merge[D-1][i/Q].sub_array[i%Q] );
   for (d=D-1; d > 0; d--) for (i=0; i < Q^d; i++)
    connect ( merge[d][i].merged_array
            , merge[d-1][i/Q].sub_array[i%Q] );
   connect ( merge[0][0].merged_array, output_array);
  }
}}
```

FIGURE 6.3. A) Prototype for specifying task and application. B) An example malleable specification for the sort application in Figure 6.2.

programming constructs such as `for` and `if-else` in order to provide enough flexibility in specifying the tasks based on the given forming parameters.

6.3. Exploration of Forming Parameter Space

To examine the merits of FORMLESS, we developed a design space exploration (DSE) scheme whose block diagram is depicted in Figure 6.4. The DSE instantiates a platform-driven task graph $G(\Phi_{opt})$ from a given FORMLESS specification by optimizing the forming vector Φ. Central to the quality of the DSE are high-level estimation algorithms for fast assessment of the throughput of a specific instance of the task graph.

6.3.1. Task Profiling. The wokload associated with a task is composed of two components: computation workload and communication-induced workload. Since tasks are defined parametrically, their computation workload depends on the values of the relevant

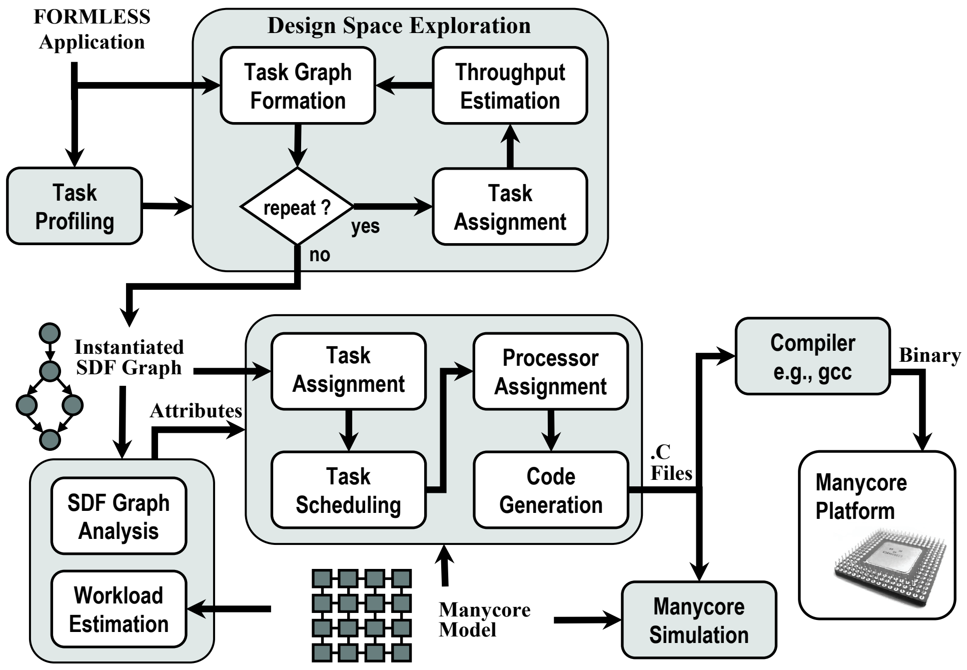


FIGURE 6.4. Design space exploration for platform-driven instantiation of a FORMLESS specification.

forming parameters. In addition, computation workload is inherently input-dependent, due to the strong dependency of the tasks' control flow with their input data. For example, the runtime of the quicksort algorithm on a list depends on the ordering of the numbers in the list. The communication-induced workload exists if some of the producers (consumers) of the data consumed (produced) by the task are assigned to a different processor.

We take an empirical approach to characterization of computation workload. We measure the execution latency of several instances of the tasks (based on the forming parameters) on the target processor. For each case, we profile the runtime for several randomly generated input streams to average out the impact of input-dependent execution times. The data is processed via regression testing to obtain latency estimates for all parameter values. Hence, for a task $\alpha(\Phi)$, the profiling data provides DSE with a computation workload W_α.

In addition, for a channel (α, α') with communication volume $N_{(\alpha,\alpha')}$, the communication-induced workload of producer and consumer tasks are analytically characterized as $W_{write} \times N_{(\alpha,\alpha')}$ and $W_{read} \times N_{(\alpha,\alpha')}$, respectively. W_{write} and W_{read} are the profiled execution latency of platform communication operations.

6.3.2. Task Graph Formation. Formation of a task graph is essentially assignment of valid values to the forming parameters. Any such assignment implies a specific instantiation, which can be passed onto subsequent stages for quality estimation. Our current DSE implementation exhausts the space of forming vector parameters by enumeration, due to the manageable size of the solution space in our testcases, and quickness of subsequent solution quality estimation. In principle, high-level quality estimations can analyze performance bottlenecks to provide feedback and to guide the process of value assignment to forming set parameters. Note that our primary objective in this chapter is to demonstrate the scalability of malleable specifications, and not development of a sophisticated DSE.

6.3.3. Task Assignment. Task assignment is a prerequisite to application throughput estimation, and quantifying the suitability of a candidate task graph for a target platform. Tasks' computations should be distributed among processors as evenly as possible while inter-processor communication is judiciously minimized. This can be modeled as a graph partitioning problem, in which a graph $G(V, E)$ is cut into a number of subgraphs $G_k(V_k, E_k)$, one for each processor p_k. We employ METIS graph partitioning package [**KK98**] for this purpose because our primary focus is to quickly generate solutions to enable integration within the iterative DSE flow. Every vertex (task) $\alpha \in V$ is assigned a weight W_α which denotes its computation workload, and every edge (α, α') is assigned a weight of $N_{(\alpha,\alpha')}$ which denotes its communication volume.

Normally, the onchip network of a manycore system allows all processors to communicate to one another. However, in some cases only neighbor processors may communicate due limited communication resources in the network architecture. In such a case, the in-degree and out-degree of partitions need to be limited, i.e., the number of processors that a processor reads from or write to should be limited. Otherwise, the processor assignment step is guaranteed *not* to be able to map the task assignment solution to the target manycore system.

We consider a limit D^{max} as a constraint on the in-degree and out-degree of every partition. The partitioning solution from METIS needs to be modified to respect this constraint, if needed. Let $D_{in,k}$ and $D_{out,k}$ denote the in-degree and out-degree of partition k, respectively. We employ the following method in adjusting the task assignment solution such that

$$\forall 1 \leq k \leq P \ : \ D_{in,k} \leq D^{max} \text{ and } D_{out,k} \leq D^{max} \tag{6.8}$$

METIS solution is iteratively modified by moving one task at a time from one partition to another. In each iteration, first a list of candidate moves are constructed and then the best candidate is selected. The details involved in each iteration are as follows. The partitions that violate the constraints are added into a black list. A cut edge $e(v,u) \in C$ from task v in a black listed partition $p(v)$ to task u in another partition $p(u)$, forms a candidate move, in that task v is moved from partition $p(v)$ to $p(u)$. Next, the best move is selected from the candidate list. The criteria for selection is to assume the move is applied, calculate a cost vector, and then select the move with the least cost vector. The elements in the cost vector are: 1) the number of processors that violate the constraints, 2) $\sum_{1 \leq k \leq P} \min\{0, D_{in,k} - D^{max}\} + \min\{0, D_{out,k} - D^{max}\}$, which is a more quantitative measure of how much the constraints are violated, 3) the workload of maximally loaded partition, 4) sum of the workloads of the three maximally loaded partitions, and 5) sum of the workloads of all partitions. A cost vector $x = (x_1, x_2, x_3, x_4, x_5)$ is smaller than y if and only if $x_1 < y_1$, or, $x_1 = y_1$ and $x_2 < y_2$, or, $x_1 = y_1$ and $x_2 = y_2$ and $x_3 < y_3$, etc. After the move with the least cost selected, it is applied to the task assignment solution, and the entire procedure repeats. This iterative improvement is repeated 100 times, upon which, if the solution still violates the constraint, the corresponding forming vector which formed the task graph is discarded.

6.3.4. Throughput Estimation. For typical FIFO channels with small latency (relative to processors' execution period), the communication overhead only appears as communication-induced workload of write and read tasks on processors. That is, the workload of a processor can be estimated as:

$$\begin{aligned} W_p = & \sum_{\alpha \in V_p} W_\alpha \\ & + W_{read} \times \sum_{\alpha \notin V_p, \alpha' \in V_p} N_{(\alpha,\alpha')} \\ & + W_{write} \times \sum_{\alpha \in V_p, \alpha' \notin V_p} N_{(\alpha,\alpha')} \end{aligned} \tag{6.9}$$

where W_{read} and W_{write} denote the execution latency of platform read and write system calls. The last two terms indicate communication-induced workload on p. We use workload of the slowest processor to estimate the throughput. Formally

$$Throughput = 1 \div \max_{1 \leq p \leq P} W_p \tag{6.10}$$

For a given task assignment, throughput of a candidate solution depends on the buffer sizes of the platform FIFO channels [**SGB08**], as well as the firing schedule of the tasks that are assigned to the same processor. Note that the above equation only provides a quick estimate for the DSE. In chapter 5, we proposed a method which overlaps different iterations of the application execution, and hence, reorders the execution of tasks such that throughput is equal to the above estimate for systems with large-enough FIFO buffers. Intuitively, large buffers disentangle the steady state execution of processors at which point, the throughput of the system is determined by the slowest processor.

For purpose of DSE high-level estimations, we assume the interconnection network has enough buffering capacity to disentangle steady state execution of processors. However, the communication-induced workload is accurately modeled as discussed above. Note that we accurately simulate the impact of interconnect limited buffer size in our final experimental evaluations, which are performed using synthesized software from FORMLESS models (Section 6.4). The buffers are only assumed to be large during DSE to enable fast estimation in the exploration phase.

6.4. Empirical Evaluation

6.4.1. Application Case Studies. To demonstrate the merits of our idea, we experiment with low-density parity check (LDPC), advanced encryption standard (AES), fast fourier transform (FFT), parallel merge sort (SORT) and matrix multiplication (MMUL) applications. Appendix A presents the code for malleable specification of the applications based on the following discussion.

Low-Density Parity Check: A regular LDPC code is characterized by an $M \times N$ parity check matrix, called the H matrix. N defines the code length and M is the number of parity-check constraints on the code (Figure 6.5.A). Based on matrix H, a Tanner graph

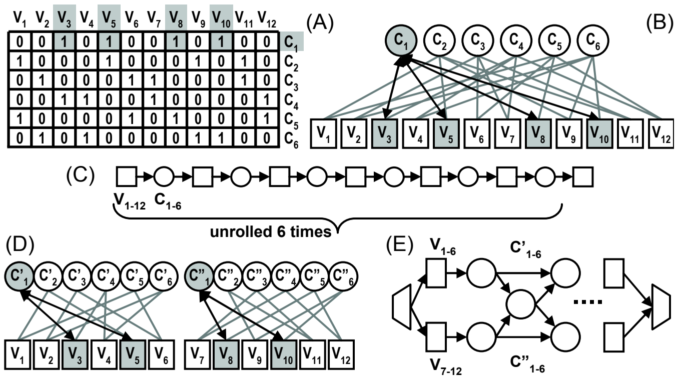


FIGURE 6.5. LDPC application: A) Sample H matrix. B) Tanner graph. C) Task graph. Row-Split LDPC based on [**MTB09, Moh10**], split factor $\phi = 2$: D) Tanner graph. E) Task graph.

is defined which has M check nodes and N variable nodes. Each check node C_i corresponds to row i in H and each variable node V_j corresponds to column j. A check node C_i is connected to V_j if H_{ij} is one (Figure 6.5.B). The input data is fed to the variable nodes, and after processing goes to the check nodes and again back to the variable nodes. This process repeats R times, where R depends on the specific application of the LDPC code. In practice, the H matrix has hundreds or thousands of rows and columns, and therefore, given the complexity of edges in the Tanner graph, we decided not to use this graph as the task graph for software implementation. In fact, direct hardware implementation of the Tanner graph is also not desired because a huge portion of the chip area would be wasted only for routing resources [**MTB09**].

We construct the task graph as the following. The variable and check nodes are collapsed into single nodes, and subsequently, the graph is unrolled R times (Figure 6.5.C). We experiment with the LDPC code used in 10GBASE-T standard, where the matrix size is 384×2048 and $R = 6$.

In order to have a malleable specification, we decided to employ the Row-Split method which is a low-complexity message passing LDPC algorithm and is originally developed for hardware implementation [**MTB09, Moh10**]. In this method, in order to reduce the complexity of the edges, the Tanner graph is generated while the rows are split by a factor of $\phi = 2$, 4, 8 or 16. As shown in Figure 6.5.D for $\phi = 2$, the variable nodes are divided into $\phi = 2$ groups, $V_1, \ldots, V_{\frac{N}{2}}$ and $V_{\frac{N}{2}}, \ldots, V_N$, and each check node C_i is split into $\phi = 2$ nodes C'_i and C''_i. The corresponding task graph is shown in Figure 6.5.E, where additional

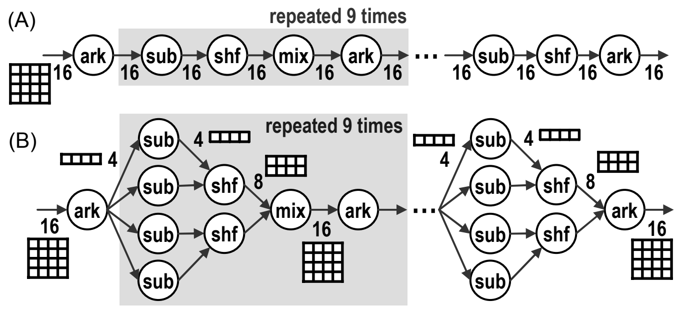


FIGURE 6.6. AES application: A) $\Phi = (1, 1, 1, 1)$ B) $\Phi = (4, 2, 1, 1)$.

synchronization nodes are required for the check nodes. For further details on the Row-Split method refer to [**MTB09, Moh10**].

Advanced Encryption Standard: The AES is a symmetric encryption/decryption application which performs a few rounds of transformations on an stream of 128-bit data (4×4 array of bytes). The number of rounds depends on the length of the key which is 10 for 128-bit keys. As shown in Figure 6.6.A, the task graph for the AES cipher consists of four basic tasks called `sub`, `shf`, `mix` and `ark`. Task `sub` is a nonlinear byte substitution which replaces each byte with another byte according to a precomputed substitution box. In `shf`, every row r in the 4×4 array is cyclically shifted by r bytes to the left. Task `mix` views each column as a polynomial x, and calculates modulo $x^4 + 1$. Task `ark` adds a round key to all bytes in the array using XOR operation. The round keys are precomputed and are different for each of the 10 rounds.

Therefore, tasks `sub` and `ark` can be parallelized over all elements of the array, and task `shf` only over the four rows, and task `mix` only over the four columns. We decided to construct the FORMLESS task graph with four parameters. ϕ_1, ϕ_2 and ϕ_4 control the number of rows that the array is divided into for the `sub`, `shf` and `ark` tasks. Parameter ϕ_3 controls the number of columns that the array is divided into for the `mix` task. For example, the task graph of Figure 6.6.B is formed by $\Phi = (4, 2, 1, 1)$.

Fast Fourier Transform: Fourier tansform of an input array is an array of the same size. Fast Fourier Transform (FFT), an efficient algorithm for this computation, is performed using a number of basic butterfly tasks connected in a dataflow network. The basic butterfly operation calculates Fourier of two inputs and is called a radix-2 butterfly. In

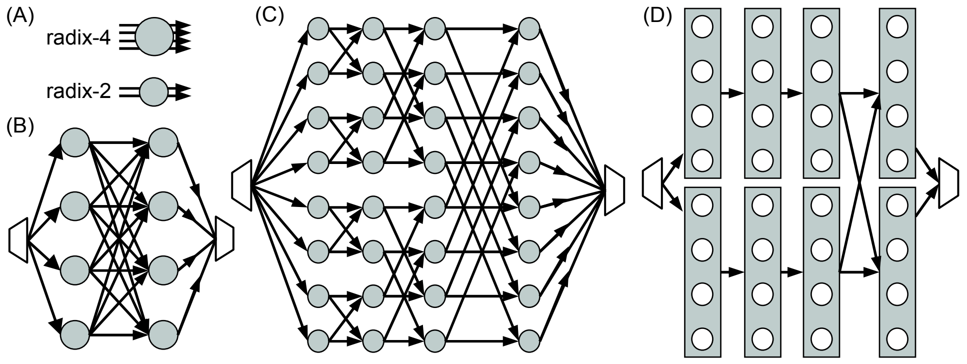


FIGURE 6.7. A) Radix-2 and radix-4 butterfly tasks. B) 16-point FFT application with radix-4 butterfly tasks. C) The same FFT computed with radix-2. D) Radix-2 task graph condensed with a factor of 4.

general, however, FFT can be calculated using butterfly operations with radices other than 2, although typically powers of 2 are used.

An N-point radix-r FFT computes the Fourier transform of an array of size N using a dataflow network of radix-r butterfly tasks. This network is organized in $\log_r^N$ stages each containing $\frac{N}{r}$ butterfly operations. Figure 6.7.B shows the structure of the dataflow network for a 16-point FFT application using radix-4 butterfly tasks. Figure 6.7.C shows the same computation performed using radix-2 butterflies. Since the computation of FFT is independent of the choice of radix, we define our FORMLESS model for FFT based on a forming parameter ϕ_1 which is the radix. The radix determines the structure of the task graph as well as inter-task data communication rates. We also define a second forming parameter ϕ_2 in order to condense the task graph as the following. If $\phi_2 = 1$, the task graph is not condensed. Otherwise, in each column of the FFT task graph, ϕ_2 butterfly tasks are grouped together into one larger task. The condensed version of the task graph in Figure 6.7.C, is shown in Figure 6.7.D, where $\phi_2 = 4$.

Parallel Merge Sort: Merge sort application divides the input array into a number of segments, sorts each segments, and iteratively merges the sorted segments into larger arrays to produce the final output. Specifically, `scatter` tasks divide the input among a number of `sort` tasks, and the sorted results are merged back by a series of `merge` tasks (Figure 6.8).

Conceptually, the input array can be split into any number of segments. Another degree of freedom in specifying the algorithm is the number of sorted segments that are combined

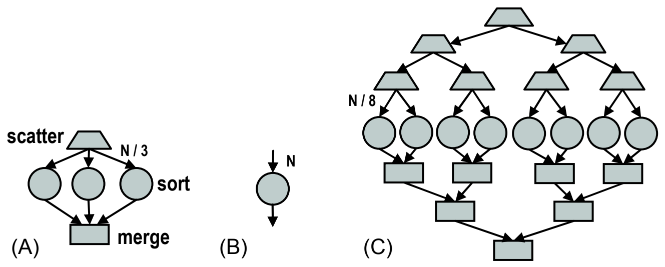


FIGURE 6.8. Parallel merge sort application constructed with A) $\Phi = (3, 3)$, B) $\Phi = (1, 1)$ and C) $\Phi = (8, 2)$.

together by a merge task. The algorithm for merging more than two input streams is a straight forward extension of merging two inputs. Clearly, in specifying the computations of the merge task, one has to consider the parametric number of input streams to the task. Therefore, one can use the forming vector $\Phi = (\phi_1, \phi_2)$ to cast the task graph in FORMLESS model, where ϕ_1 refers to the number of sort tasks, and ϕ_2 controls the fanout and fanin degree of the scatter and merge actors. If $\phi_2 = 1$, there is no scatter or merge actor and the task graph has only one sort actor. Hence, the value of ϕ_1 is an integer power of ϕ_2 to generate a valid task graph, i.e., $\phi_1 = \phi_2^n \ , \ n > 0$.

For example, the task graph of Figure 6.8.A is instantiated using the forming vector $\Phi = (3, 3)$, because there are three sorted arrays that are combined using one merge task. The forming vector $\Phi = (1, 1)$ would result in the task graph of Figure 6.8.B, since there is only one sort task. Figure 6.8.D is formed by forming vector $\Phi = (8, 2)$.

Matrix Multiply: To multiply two matrices A and B, rows of A should be multiplied by columns of B. Figure 6.9.A shows an example in which, element $(7, 5)$ of the result matrix C is calculated from row 7 of matrix A and column 5 of matrix B. Calculation of elements of matrix C are independent of each other, and in principle, they can be carried out concurrently. This approach, however, would require massive replication of data in matrices A and B.

The basic operation of calculating an element of matrix C from rows and columns of A and B, can be generalized to calculating block of C from multiple rows and columns of A and B. Figure 6.9.B shows an example in which, calculating the shaded block of matrix C, C_{21}, requires the data from shaded rows of A (A_2) and shaded columns of B (B_1). Adjusting the block size in C trades off the degree of concurrency among operations with the required amount of data replication and movement.

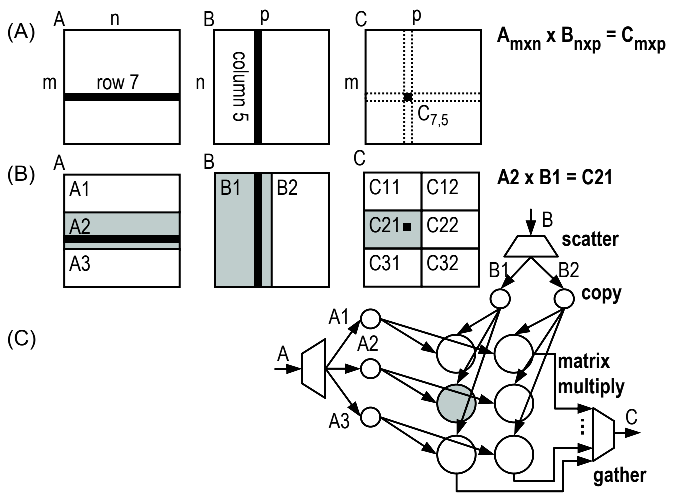


FIGURE 6.9. A) Matrix multiply. B) Parallelized matrix multiply for $\Phi = (3, 2)$. C) Task graph with $\Phi = (3, 2)$.

Note that unlike matrix addition, multiplication of matrices is not purely data parallel, because the pieces of data, i.e., sub-matrices, are required in more than one sub-matrix multiplication. In a pure data parallel application, such as matrix add, the input data can be simply scattered to many tasks and gathered back without any extra processing before and after the parallel operation.

Figure 6.9.C illustrates rigid task graph of the example in Figure 6.9.B. The `scatter` tasks divide matrix A into horizontal sub-matrices, and matrix B into two vertical sub-matrices. `Copy` tasks copy these sub-matrices to corresponding `multiply` tasks where the multiplication is performed on the sub-matrices.

Thus, there are two degrees of freedom in modeling the application as a FORMLESS task graph. Specifically, the number of row and column blocks that matrices A and B are divided into can be adjusted independently. The forming set can be defined to include the parameters $\Phi = (\phi_1, \phi_2)$, where ϕ_1 (ϕ_2) determines the number of row (column) blocks of matrix A (B). The task graph of Figure 6.9.C is formed by $\Phi = (3, 2)$. The two parameters would imply the size of blocks in matrix C that are modeled as concurrent tasks in the application graph.

Domain Δ: The domains of the forming vectors used in experimenting the above applications are shown in Figure 6.10. For example in the AES application, each of the four forming parameters can be 1, 2 or 4.

Benchmark	Vector Φ	$\Delta =$ Domain of vector Φ	$\|\Delta\|$
AES	$(\phi_1, \ldots, \phi_4)$	$\delta_1 = \delta_2 = \delta_3 = \delta_4 = \{1, 2, 4\}$	81
FFT	(ϕ_1, ϕ_2)	$\delta_1 = \{2, 4, 8, 16\}, \delta_2 = \{1, 2, 4, \ldots, 128\}$	32
SORT	(ϕ_1, ϕ_2)	$\delta_1 = \{1, \ldots, 100\}, \delta_2 = \{1, \ldots, 10\}, \phi_1 = \phi_2^n$	26
MMUL	(ϕ_1, ϕ_2)	$\delta_1 = \delta_2 = \{1, \ldots, 16\}$	256
LDPC	(ϕ_1)	$\delta_1 = \{1, 2, 4, 8, 16\}$	5

FIGURE 6.10. Domain of the forming parameters in our benchmark applications. In the SORT benchmark, there are only a total of 26 cases since not all combinations of $\phi_1 \in \delta_1$ and $\phi_2 \in \delta_2$ are in the domain.

6.4.2. Experiment Setup. We implemented both FORMLESS design space exploration and baseline software synthesis schemes (Figure 6.4). For a given number of processors, P, within the range of 1 to 100, an optimized task graph $G(\Phi_{opt})$ is constructed, and subsequently, parallel software modules (separate `.C` files) are synthesized for this task graph. In synthesizing the parallel software modules, the task assignment step is as described in Section 6.3, the task scheduling is a simple topological traversal of the task graph following the iteration overlapping method in Chapter 5, and the processor assignment step is manual.

We consider the following FPGA-prototyped multiprocessor system for throughput measurement of the synthesized software modules. Each processor is an Altera NiosII/f core with 8KB instruction cache and 32KB data cache. The communication network is a mesh which connects the neighbor processors with FIFO channels of depth 1024. The processors use a shared DDR2-800 memory, but they only access their own region in this memory, i.e., they communicate only through the FIFO channels. The compiler is `nios-gcc` in Altera NiosII IDE with optimization flag `-O2`. Due to limited FPGA capacity, we were able to implement the above architecture with up to 8 cores. For more number of processors, we estimated the throughput using SEAM, i.e., the abstract model discussed in chapter 4. In constructing the abstract model we used the estimated workload (Section 6.3).

6.4.3. Experiment Results. The experiment results confirm that throughput of the best instantiated task graph consistently beats the throughput of any rigid task graph. Figure 6.11 presents the application throughput numbers normalized relative to single-core throughput. The black curves show the throughput values obtained through SEAM simulations from synthesized parallel implementations, and the 8 black squares show the

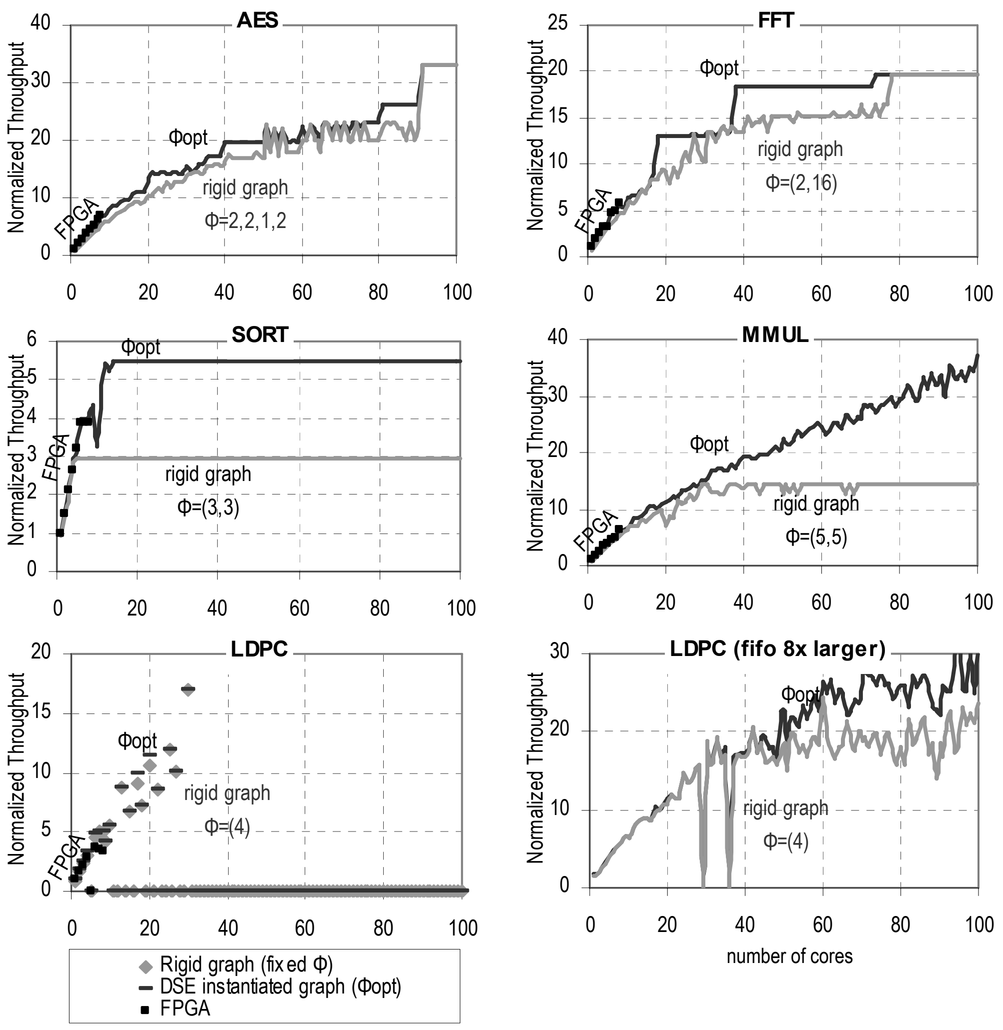


FIGURE 6.11. Application throughput on manycore platforms normalized with respect to single-core throughput. The black curve shows the throughput obtained from the DSE instantiated task graphs, i.e., $G(\Phi_{opt})$. The gray curves show the throughput of sample rigid task graphs. Deadlock is denoted with a zero throughput value in the figure.

throughput measured on the FPGA prototype for systems up to 8 processors. The gray curves show throughput of few rigid task graphs selected for our discussion. There are two LDPC charts in the figure and the reason is described later in Section 6.4.6.

The experiments confirm that rigid task graphs have a limited scope of efficient portability and scalability with respect to number of processors. For example, an LDPC rigid task graph constructed with forming vector $\Phi = (4)$ does not scale beyond 40 processors (bottom right corner in Figure 6.11). Note that a rigid task graph which scales to large number of processors does not necessarily yields the best throughput in smaller number of processors. For example, an AES rigid task graph constructed with $\Phi = (2, 2, 1, 2)$ only yields the highest throughput for 90 or more processors (top left corner in Figure 6.11). It is hard to see in the figure, but for a single processor the rigid AES task graph yields 74% throughput of the best instantiated task graph.

Similar scenarios happen in all benchmark case studies. Each forming vector Φ yields the highest throughput only for a range of targets. This result validates the effectiveness of FORMLESS in extending the scope of efficient portability and scalability with respect to number of cores.

It is interesting to see that, for example, in the matrix multiply application $\Phi = (5, 5)$ is not selected for the 25-core target. Instead, the DSE tool selected $\Phi = (6, 4)$ which has 24 multiply tasks. This forming set is not intuitive because one would normally split the multiplication workload into an array of $5 \times 5 = 25$ multiply tasks for 25 cores. The DSE tool considers the effect of smaller tasks (e.g., scatter tasks), and the communication-induced workloads as well. This again proves that an automated tool outperforms manual task graph formation. However, the DSE is able to scale performance only if the programmer has provided meaningful parallelism. For example in the SORT application, a larger value for the forming parameter ϕ_1 results in more parallel sort tasks, but the performance does not scale beyond 13 processors because the workload of the last merge stage is very large and it is not parallelized.

Figure 6.11 can also be used to determine a reasonable target size, i.e., the number of processors, for each application. For example in the AES application, more than 40 processors does not yield a throughput gain unless we have at least 50 processors.

Comparisons between the estimated throughput given by SEAM (black curves in Figure 6.11) with the throughput measured on FPGA (black squares in Figure 6.11) confirms that SEAM is relatively accurate. As we see the two values are very close to each other.

6.4.4. Improving the DSE. As mentioned before, for a given number of processors, P, within the range of 1 to 100, the DSE selects an optimum forming vector Φ_{opt} and constructs the corresponding task graph $G(\Phi_{opt})$ for software synthesis. Since the main objective in this chapter is to demonstrate the scalability of malleable specifications, and not development of a sophisticated DSE, the current DSE algorithm implements only an exhaustive search of all the possible forming vectors. However, we plan to improve DSE to judiciously search only among a small subset of possible forming vectors. In fact, some vectors are never selected by the DSE for any target size, and thus, only a portion of the vectors are enough to cover all the target sizes. In other words, the set of DSE selected forming vectors Φ_{opt} across all target sizes ($1 \leq P \leq 100$) is a small subset of Δ, the domain of Φ. For example in the AES application, size of Δ is 81 but size of this set is 13, i.e., $81 - 13 = 68$ forming vectors are never selected by the DSE for any target size.

In order to have detailed statistics on this matter, we analyzed the DSE results as the following. Forming vectors are sorted based on their frequency of selection by the

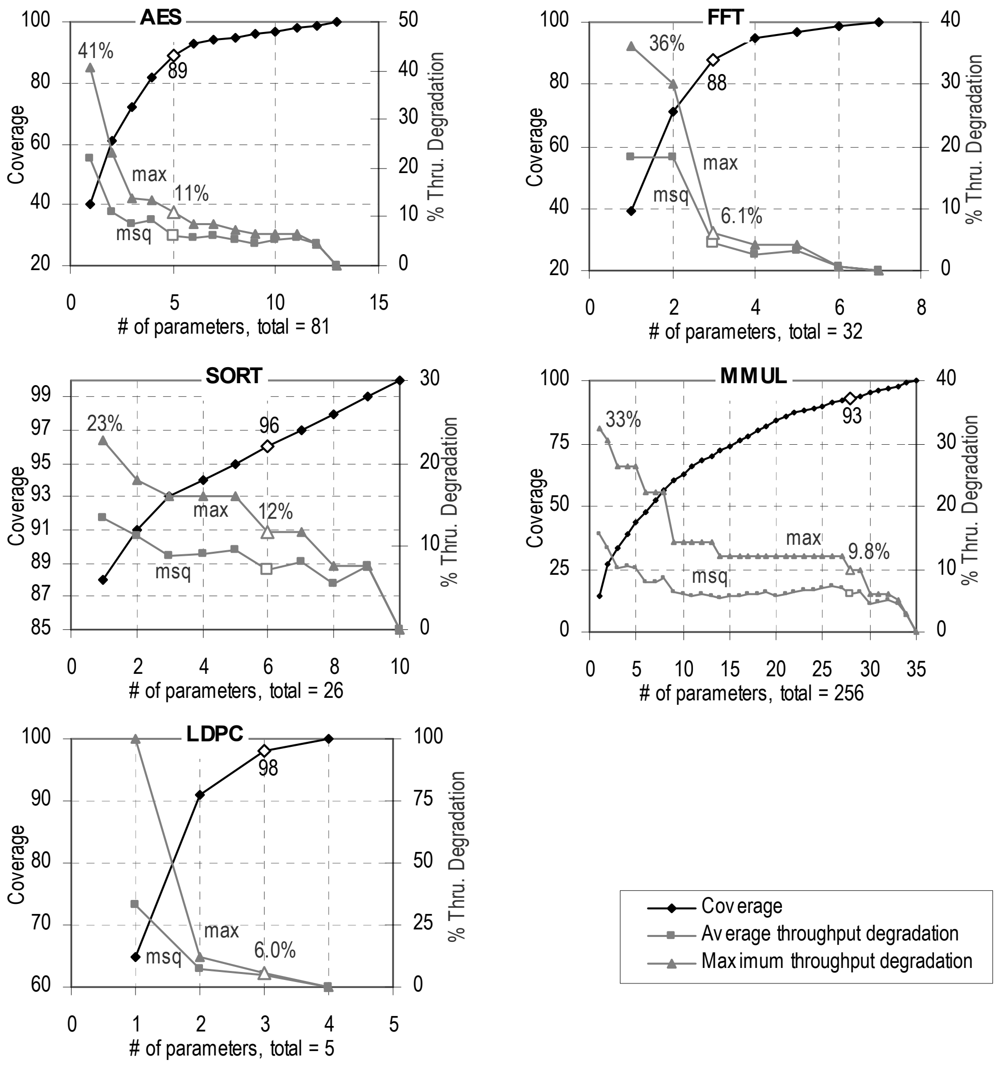


FIGURE 6.12. Coverage and throughput degradation vs the number of forming vectors (Section 6.4.4).

DSE across all target sizes. The black curves in Figure 6.12 show the cumulative selection frequency (coverage) of the sorted vectors. For example in the AES application (top left), the first vector (in the sorted order) is selected by the DSE in 40 out of 100 different target sizes. The second vector is selected in 21 target sizes, and hence, the two vectors cover $40 + 21 = 61$ target sizes. The coverage increases to 100 with more number of forming vectors. In the AES application, for example, 5 vectors are enough to cover 89 target sizes, and 13 vectors are enough to cover all target sizes.

The gray curves in Figure 6.12 show the throughput degradation as a result of considering only a subset of the forming vectors (in the above sorted order). For example in the AES application (top left), the first 5 forming vectors cover 89 target sizes, and in the other $100 - 89 = 11$ uncovered target sizes, this set of forming vectors result in a throughput degradation of 11% in the worst case and 6.2% on average (mean square).

The amount of throughput degradation decreases with more forming vectors. In order to instantiate task graphs with around 10% maximum throughput degradation, only a small number of the forming vectors are required, 15% of the vectors on average (geometric mean). See the hallow points in Figure 6.12.

6.4.5. Accuracy of SEAM. Recall from Chapter 4 that SEAM accurately models the behavior of the onchip network through RTL Verilog simulation of the network with Modelsim. However, SEAM does not model the effect of cache. Specifically, SEAM uses the workload estimates (Section 6.3) as the execution latency of tasks, and hence, small cache sizes may increase this latency and cause inaccuracy. Figure 6.13 shows the throughput measured on FPGA for different data cache sizes, along with the SEAM estimation. We observe that when data cache size is 32KB, SEAM is very accurate in all benchmarks, except LDPC which has large data sets. In LDPC application, SEAM estimations follow the same trend in throughput scaling as the FPGA measurements.

Data memory requirements of the produced codes are gathered following compilation with `nios-gcc` with `-O2` optimization flag. Minimum, average and maximum size of the data memory section of the codes are also reported in Figure 6.13. The values are measured from the compiled `.elf` binary using `nios-elf-size` program.

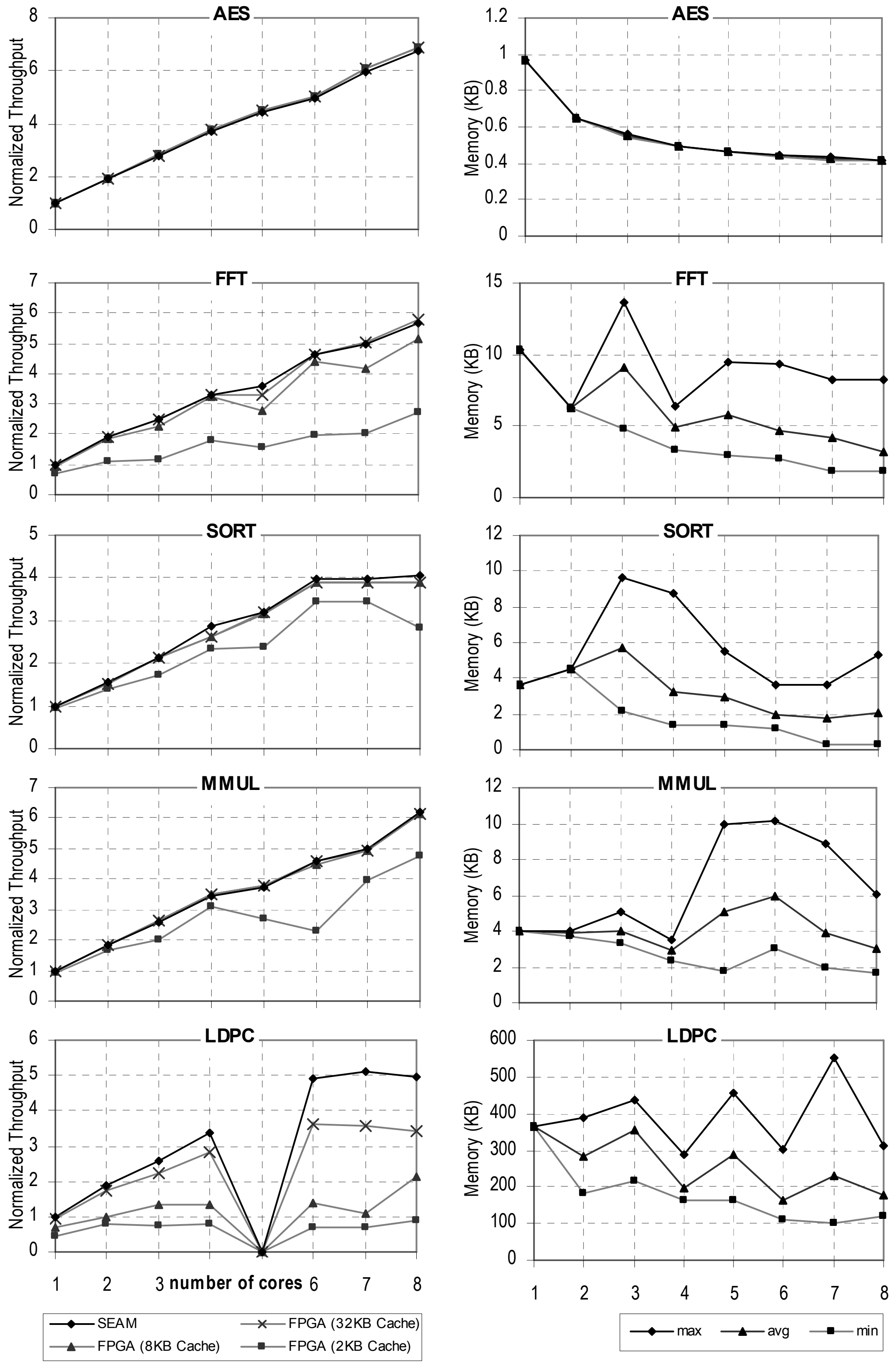


FIGURE 6.13. Application throughput normalized with respect to single-core throughput for different data cache sizes.

6.4.6. Effect of FIFO Depth. For a cut channel $e(v, u) \in C$ from task v with production rate of p to a task u with consumption rate of c, previous studies [**ALP97, SGB08**] mention that the minimum buffer size required to avoid deadlock is

$$p + c - \gcd(p, c) \tag{6.11}$$

The right column in Figure 6.14 shows the value calculated from this equation for all the benchmarks. Based on this equation, a FIFO size of 1024 is not enough to avoid deadlock in the LDPC benchmark. As shown previously in Figure 6.11 on page 102 (the bottom left chart), the LDPC benchmark has deadlock in 82 cases out of 100 when FIFO size is 1024. Deadlock is denoted with a zero throughput value in the figures. The bottom right chart in Figure 6.11 on page 102 shows the throughput when FIFO size is 8192. Deadlock happens in only 2 cases for this larger FIFO size. Further investigation revealed the reason. The task assignment algorithm creates a cyclic dependency between two processors which makes the FIFO size requirement go higher than the value in Equation 6.11.

Interestingly, we observed that in many cases FIFO sizes smaller than $p + c - \gcd(p, c)$ are enough not to reduce the throughput or cause deadlock, and in few other cases even larger FIFO sizes cause deadlock. We further investigated this matter to find the reason. The left column shows the throughput estimated by SEAM for two different FIFO sizes, one large FIFO whose size if close to the value from Equation 6.11 (dark curve in the figure), and one small FIFO (light gray points in the figure). Recall from Chapter 4 that SEAM only abstracts the local execution phases of every processor, while the behavior of the onchip network is accurately modeled through RTL Verilog simulation of the network with Modelsim.

As we see in the figure, for example, in the AES application (top right), a minimum FIFO size of 8, 16 or 32 is required to avoid deadlock based on Equation 6.11. However, we observed that a FIFO size of 2 avoids deadlock in 89 out of 100 cases, and this small FIFO size does not even reduce the throughput in 80 cases (top left figure). FFT requires a FIFO size of 128 or 256 in most cases, but we observed that 32 is enough in 92 cases. Similar scenario happens in other benchmarks as well.

This is because formulations such as Equation 6.11 [**ALP97, SGB08**] only consider the dataflow specification in its pure mathematical form and ignore the implementation

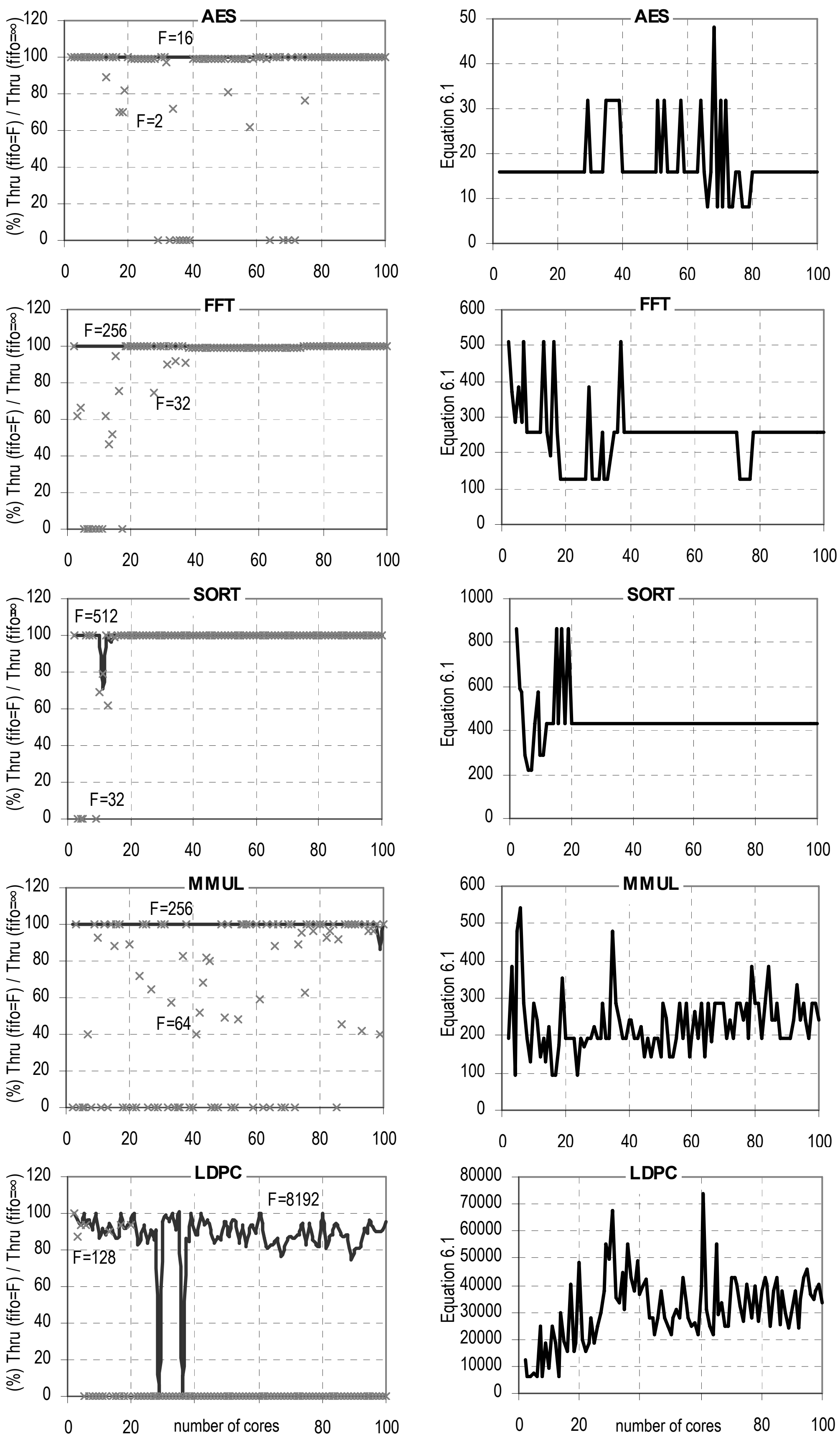


FIGURE 6.14. Left: ratio of the throughput for a specific FIFO size F, over the throughput with infinite FIFO. Right: minimum buffer sizes required to avoid deadlock, calculated from Equation 6.11.

details. The mathematical model is that a task consumes all the required tokens at once upon firing, and also produces all the output tokens at once. However, in reality the data should be transfered from the local memory of the processors to the FIFO channel, and this is normally implemented with a software loop or a hardware DMA unit which transfers the data tokens one by one, and not at once.

6.5. Related Work and Chapter Summary

Parametrized dataflow [**BB01**] and dataflow interchange format [**P$^+$08**] models primarily focus on specifications which enable different static and/or dynamic dataflow behaviors based on the parameters. We, however, focused on specifying different possible implementations of the same application behavior, in order to achieve scaling of performance with respect to the number of processors. Therefore, we were able to employ a richer set of programming constructs to specify many aspects of the task graph based on the forming parameters (Section 6.2.2). For example, as opposed to previous works [**BB01, P$^+$08**], not only the production/consumption port rates but also the number of ports for each task can be specified based on the forming parameters in FORMLESS.

StreamIt compiler [**Gor10**] automatically detects stateless filters (data-parallel tasks) and judiciously parallelizes them in order to achieve better workload balance and hence scaling of performance. This approach provides some level of malleability, but it is limited to data-parallel tasks because it fully relies on the *compiler's* ability to detect malleable sections in the application.

In CUDA, scaling of performance is achieved by specifying the application with as much parallelism as practically possible. At runtime, an online scheduler has access to a pool of threads from which the non-blocked threads are selected and executed on the available processors [**N$^+$08**]. This enables scaling of performance to newer devices with larger number of processors. However, performance optimization for a specific target GPU device fully relies on the *programmer* to optimally specify the application, e.g., the number of blocks per grid and the number of threads per block [**Cud11**].

This chapter proposed a FORMLESS model in which the *programmer* explicitly provides a malleable specification, and the *complier* optimizations select the best task graph based on the malleable specification.

CHAPTER 7

Conclusion and Future Work

Abundant parallelism and predictable data accesses in streaming applications provide the opportunity to efficiently map high-level streaming specifications to target manycore platforms. Chapters 2 to 5 presented our contributions on efficient optimization and estimation techniques required for automated synthesis of streaming software. Different optimization steps involved in automated software synthesis of stream applications affect one another, a summary of which is the following.

- Task assignment judiciously partitions the task graph to achieve a balanced workload while avoiding excessive communication overhead. Restricting the task assignment to convex partitions prohibit many solutions which may potentially have a better workload balance. However, this restriction simplifies the task scheduling process by avoiding cyclic dependencies between the processors. On the other hand, allowing non-convex partitions may improve the workload balance but requires iteration overlapping techniques in scheduling of the tasks in order to achieve the expected throughput. Otherwise the cyclic dependencies which did not exist in the task graph and were created as a result of non-convex partitions may significantly reduce the throughput. To this point, one may decide to favor non-convex partitions. Nevertheless, iteration overlapping requires more buffer memory, which may reduce the throughput since hardware or software managed cache units may need to spill data in and out of the slow main memory more often. The greater impact, however, is on onchip communications. Buffer memory required between two tasks assigned to separate processors is also increased as a result of iteration overlapping. This may reduce the throughput or even worse cause deadlock if enough buffering is not available in the architecture. Further investigation into the interrelated effects of task assignment, iteration overlapping and inter-processor buffering capacity is required to enable safe usage of non-convex partitions and iteration overlapping.

- Processor assignment judiciously assigns the logical processors (partitions from task assignment) to physical processors, in order to conform to the limitations of onchip communication resources and/or avoid heavy long distance communications and network congestion. Separation of task assignment and processor assignment steps simplifies the problem by dividing it into two subproblems. However, task assignment should be aware of architecture limitations and produce partitions that are digestable by the processor assignment. For example, in mesh architectures with limited connections to neighbor processors, task assignment should at least limit the in-degree and out-degree of the produced partitons.

- Embedded manycore systems often have limited local memory which may need to be considered as a constraint during the optimization steps such as task assignment or task scheduling. Our experience shows that most of the optimizations following the task assignment step have only a minimal impact on the instruction memory requirement of the parallel software modules. However, data memory requirement often varies by a great degree in every step. Violating the architecture limitation on instruction memory often renders the solution infeasible, while violating the limitation on data memory can often be mitigated by spilling data to main memory at a performance cost. Hence, it is often enough to consider the constraint on instruction memory requirements only during the task assignment step. However, an estimate of the impact on data memory requirement on future optimization steps should ideally be considered in earlier steps, or alternatively, later steps may adjust the solution, at possible performance costs, to meet the constraints on or to optimize the data memory requirement.

- In addition to the workload of processors, inter-processor data communications as well as processor-memory data communications contribute to the power consumption of manycore platforms. Therefore, any power constraint may be considered as part of task assignment and processor assignment steps, which have direct impact on workloads and communications. In other words, power consumption can be estimated based on the workloads and communications. However, in order to have a more precise control, power modeling of tasks based on their high-level source code, and data communications based on their rates or volume are required.

Gaining insight from our previous works, we concluded that achieving portability and scalability across a wide range of manycore platforms requires not only efficient optimization techniques, but also right abstractions for both the high-level application specification model and the manycore platform model. Chapter 6 presented a preliminary evaluation of the opportunity that a higher-order malleable dataflow model such as FORMLESS can provide to software synthesis.

It is a costly and time-consuming practice for the programmer to go through the cycle of specifying the application, perform automated software synthesis, find bottlenecks and other limitations, and re-specify the application. On the other hand, it is very difficult for the compiler to infer malleable portions of the application specification and adjust that automatically without any help from the programmer. The idea behind FORMLESS is that the programmer should *help* the compiler to automatically find the best task graph specification by provding a malleable higher-order specification. Future directions in advancing the FORMLESS model are the following.

- Map-reduce applications, such as sort networks, normally have a tree-like graph structure which can potentially be formed in infinite possible ways, i.e., infinite possible tree structures. Each node in the tree may have a different type and number of sub-trees based on its forming parameters. The programmer may specify such malleable map-reduce network in a recursive manner. The degree of recursion on each node would be analogous to the depth of its sub-tree. Therefore, both the malleable specification model and the design space exploration should support forming parameters that may exist only as a result of specific *values* of other forming parameters. In other words, FORMLESS should support forming vectors of variable size.

- Currently exploration of the forming parameter space is exhaustive. There are two alternatives for development of a sophisticated exploration in the future. The first option would be to start from an initial forming vector and iteratively improve the solution by guiding the algorithm to target throughput bottlenecks in restructuring the task graph. The second option would be to employ the mathematical formulation of the workload estimates based on the forming parameters, and prune portions of the parameter space which are guaranteed to yield inferior solutions.

- Another direction for future work is development of FORMLESS-aware optimization steps especially task assignment and processor assignment. A quick exploration pass may provide a reasonable task graph to software synthesis steps. However, it is also possible to keep the malleability throughout the software synthesis and change the graph structure based on the bottlenecks or limitations faced in each step.
- Scatter and gather of data among the tasks should be separated from the task graph structure, even in the formed (rigid) graph instantiated by the parameter space exploration. Such data dependences are simple enough to be automatically implemented by software synthesis steps based on a given situation. For example, scattering of data among a large number of consumer tasks may be implemented with a single scatter task, a tree network of scatter tasks, or a chain of scatter tasks. Specific implementation of such operations should be left to software synthesis.

Bibliography

[ABC$^+$06] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The landscape of parallel computing research: A view from berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[AED07] J. H. Ahn, M. Erez, and W. J. Dally, *Tradeoff between data-, instruction-, and thread-level parallelism in stream processors*, Proceedings of the International Conference on Supercomputing (2007), 126–137.

[AL07] A. Agarwal and M. Levy, *The kill rule for multicore*, Proceedings of the 44th annual Design Automation Conference, DAC '07, 2007, pp. 750–753.

[ALP97] M. Ade, R. Lauwereins, and J. Peperstraete, *Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets*, Design Automation Conference, 1997. Proceedings of the 34th, jun 1997, pp. 64 –69.

[BB01] B. Bhattacharya and S. Bhattacharyya, *Parameterized dataflow modeling for DSP systems*, IEEE Transactions on Signal Processing **49** (2001), no. 10, 2408–2421.

[BEA$^+$08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, *Tile64 - processor: A 64-core soc with mesh interconnect*, Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International, feb. 2008, pp. 88 –598.

[BGS94] D. F. Bacon, S. L. Graham, and O. J. Sharp, *Compiler transformations for high-performance computing*, ACM Comput. Surv. **26** (1994), 345–420.

[BHH$^+$07] R. I. Bahar, D. Hammerstrom, J. Harlow, W. H. J. Jr., C. Lau, D. Marculescu, A. Orailoglu, and M. Pedram, *Architectures for silicon nanoelectronics and beyond*, Computer **40** (2007), no. 1, 25 –33.

[BLM96] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software synthesis from dataflow graphs*, Kluwer Academic Publishers, 1996.

[BML99] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Synthesis of embedded software from synchronous dataflow specifications*, J. VLSI Signal Process. Syst. **21** (1999), 151–166.

[Bor07] S. Borkar, *Thousand core chips: a technology perspective*, Proceedings of the 44th annual Design Automation Conference, DAC '07, 2007, pp. 746–749.

[BWH$^+$03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, *Metropolis: an integrated electronic system design environment*, Computer **36** (2003), no. 4, 45 – 52.

[BYLN09] E. D. Berger, T. Yang, T. Liu, and G. Novark, *Grace: safe multithreaded programming for C/C++*, SIGPLAN Not. **44** (2009), 81–96.

[Cat06] J. A. Cataldo, *The power of higher-order composition languages in system design*, Ph.D. thesis, University of California, Berkeley, 2006.

[CGHP04] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet, *Towards a higher-order synchronous data-flow language*, International Conference on Embedded Software (2004), 230–239.

[CHJ07] J. Cong, G. Han, and W. Jiang, *Synthesis of an application-specific soft multiprocessor system*, Proceedings of the International Symposium on Field Programmable Gate Arrays (2007).

[CLRS01a] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, ch. 24.2, 2001.

[CLRS01b] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, ch. 27.5, 2001.

[Cud11] Cuda, *CUDA C best practices guide*, ch. 4.4, March 2011.

[dK02] E. A. de Kock, *Multiprocessor mapping of process networks: a JPEG decoding case study*, Proceedings of the International Symposium on System Synthesis (2002), 68–73.

[Eat05] W. Eatherton, *The push of network processing to the top of the pyramid*, Keynote Presentation at Symposium on Architectures for Networking and Communications Systems (2005).

[EEP06] C. Erbas, S. C. Erbas, and A. D. Pimentel, *Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design*, IEEE Transactions on Evolutionary Computation **10** (2006), no. 3, 358–374.

[FHHG10] M. H. Foroozannejad, M. Hashemi, T. L. Hodges, and S. Ghiasi, *Look into details: the benefits of fine-grain streaming buffer analysis*, Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems, LCTES '10, 2010, pp. 27–36.

[FHHG11] M. H. Foroozannejad, T. L. Hodges, M. Hashemi, and S. Ghiasi, *Post-scheduling buffer management tradeoffs insynthesis of streaming applications*, ACM Trans. Embed. Comput. Syst. (2011).

[FRRJ07] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, *Energy-optimizing source code transformations for operating system-driven embedded software*, ACM Trans. Embed. Comput. Syst. **7** (2007), 2:1–2:26.

[G$^+$02] M. I. Gordon et al., *A stream compiler for communication-exposed architectures*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (2002).

[GB04] M. Geilen and T. Basten, *Reactive process networks*, Proceedings of the 4th ACM international conference on Embedded software, EMSOFT '04, 2004, pp. 137–146.

[GGB$^+$06] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. D. Theelen, M. R. Mousavi, and S. Stuijk, *Liveness and boundedness of synchronous data flow graphs*, Proceedings of the Formal Methods in Computer Aided Design (2006), 68–75.

[GGS$^+$06] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. Bekooij, *Throughput analysis of synchronous data flow graphs*, International Conference on Application of Concurrency to System Design, 2006, pp. 25–36.

[GJ90] M. R. Garey and D. S. Johnson, *Computers and intractability; a guide to the theory of np-completeness*, W. H. Freeman, 1990.

[GLGN$^+$08] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, *Parallel computing experiences with cuda*, Micro, IEEE **28** (2008), no. 4, 13 –27.

[Gor10] M. Gordon, *Compiler techniques for scalable performance of stream programs on multicore architectures*, Ph.D. thesis, Massachusetts Institute of Technology, 2010.

[GTA06] M. I. Gordon, W. Thies, and S. Amarasinghe, *Exploiting coarse-grained task, data, and pipeline parallelism in stream programs*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (2006).

[HCK$^+$09] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, *Flextream: Adaptive compilation of streaming applications for heterogeneous architectures*, Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, 2009, pp. 214–223.

[HCW$^+$10] A. H. Hormati, Y. Choi, M. Woh, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, *Macross: macro-simdization of streaming applications*, Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, ASPLOS '10, 2010, pp. 285–296.

[HFC$^+$11] M. Hashemi, M. H. Foroozannejad, L. Chen, C. Etzel, and S. Ghiasi, *The power of formless dataflow specifications in scalable programming of embedded manycore platforms*, Submitted to ACM Transactions on Design Automation of Electronic Systems (2011).

[HG08] M. Hashemi and S. Ghiasi, *Exact and approximate task assignment algorithms for pipelined software synthesis*, Proceedings of the conference on Design, automation and test in Europe, DATE '08, 2008, pp. 746–751.

[HG09] M. Hashemi and S. Ghiasi, *Throughput-driven synthesis of embedded software for pipelined execution on multicore architectures*, ACM Trans. Embed. Comput. Syst. **8** (2009), 11:1–11:35.

[HG10] M. Hashemi and S. Ghiasi, *Versatile task assignment for heterogeneous soft dual-processor platforms*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **29** (2010), no. 3, 414 –425.

[HG11a] M. Hashemi and S. Ghiasi, *Iteration overlapping: Exploiting throughput-memory tradeoffs in streaming applications*, Submitted to ACM Trans. Embed. Comput. Syst. (2011).

[HG11b] M. Hashemi and S. Ghiasi, *Formless: Scalable and productive utilization of embedded many-cores in streaming applications*, Cool Work-In-Progress Poster Session, Proceedings of the Design Automation Conference, 2011.

[HHG07] P.-K. Huang, M. Hashemi, and S. Ghiasi, *Joint throughput and energy optimization for pipelined execution of embedded streaming applications*, Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '07, 2007, pp. 137–139.

[HHG08] P.-K. Huang, M. Hashemi, and S. Ghiasi, *System-level performance estimation for application-specific mpsoc interconnect synthesis*, Proceedings of the 2008 Symposium on Application Specific Processors, 2008, pp. 95–100.

[HM05] J. Hu and R. Marculescu, *Energy- and performance-aware mapping for regular NoC architectures*, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems **24** (2005), no. 4.

[HS06] T. A. Henzinger and J. Sifakis, *The embedded systems design challenge*, International Symposium on Formal Methods (2006), 1–15.

[IHK04] C. Im, S. Ha, and H. Kim, *Dynamic voltage scheduling with buffers in low-power multimedia applications*, ACM Trans. Embed. Comput. Syst. **3** (2004), 686–705.

[JPSN09] P. Joshi, C.-S. Park, K. Sen, and M. Naik, *A randomized dynamic program analysis technique for detecting real deadlocks*, Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, PLDI '09, 2009, pp. 110–120.

[KDH$^+$05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, *Introduction to the cell multiprocessor*, IBM Journal of Research and Development **49** (2005), no. 4.5, 589 –604.

[KK98] G. Karypis and V. Kumar, *METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system*, Tech. report, Department of Computer Science. University of Minnesota, Minneapolis, 1998.

[KM08] M. Kudlur and S. Mahlke, *Orchestrating the execution of stream programs on multicore platforms*, Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, PLDI '08, 2008, pp. 114–124.

[KMB07] M.-Y. Ko, P. K. Murthy, and S. S. Bhattacharyya, *Beyond single-appearance schedules: Efficient dsp software synthesis using nested procedure calls*, ACM Trans. Embed. Comput. Syst. **6** (2007).

[KTA03] M. Karczmarek, W. Thies, and S. Amarasinghe, *Phased scheduling of stream programs*, Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, LCTES '03, 2003, pp. 103–112.

[LBDM02] Y.-H. Lu, L. Benini, and G. De Micheli, *Dynamic frequency scaling with buffer insertion for mixed workloads*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **21** (2002), no. 11, 1284 – 1305.

[Lee05] E. A. Lee, *Building unreliable systems out of reliable components: The real time story*, Tech. Report UCB/EECS-2005-5, EECS Department, University of California, Berkeley, 2005.

[Lee06] E. Lee, *The problem with threads*, Computer **39** (2006), no. 5, 33 – 42.

[LGX$^+$09] W. Liu, Z. Gu, J. Xu, Y. Wang, and M. Yuan, *An efficient technique for analysis of minimal buffer requirements of synchronous dataflow graphs with model checking*, Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '09, 2009, pp. 61–70.

[LKJ$^+$08] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, *Facsim: a fast and cycle-accurate architecture simulator for embedded systems*, Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems, LCTES '08, 2008, pp. 89–100.

[LM87a] E. A. Lee and D. G. Messerschmitt, *Synchronous data flow*, Proceedings of the IEEE **75** (1987), no. 9, 1235–1245.

[LM87b] E. A. Lee and D. G. Messerschmitt, *Static scheduling of synchronous data flow programs for digital signal processing*, Computers, IEEE Transactions on **C-36** (1987), no. 1, 24 –35.

[LP95] E. A. Lee and T. M. Parks, *Dataflow process networks*, Proceedings of the IEEE **83** (1995), no. 5, 773–801.

[MB04] P. K. Murthy and S. S. Bhattacharyya, *Buffer merging a powerful technique for reducing memory requirements of synchronous dataflow specifications*, ACM Trans. Des. Autom. Electron. Syst. **9** (2004), 212–237.

[Mee06] Meeting, *Joint United States-European Union-TEKES workshop: Long term challenges in high confidence composable embedded systems*, June 2006, http://www.truststc.org/euus/wiki/Euus/HelsinkiMeeting.

[Mic] MicroBlaze, www.xilinx.com/microblaze.

[Mic94] G. D. Micheli, *Synthesis and optimization of digital circuits*, McGraw-Hill, 1994.

[MK08] I. Moulitsas and G. Karypis, *Architecture aware partitioning algorithms*, Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing, ICA3PP '08, 2008, pp. 42–53.

[MKWS07] S. Meijer, B. Kienhuis, J. Walters, and D. Snuijf, *Automatic partitioning and mapping of stream-based applications onto the intel IXP network processor*, Proceedings of the International Workshop on Software and Compilers for Embedded Systems (2007), 23–30.

[Moh10] T. Mohsenin, *Algorithms and architectures for efficient low density parity check (LDPC) decoder hardware*, Ph.D. thesis, University of California, Davis, 2010.

[MTB09] T. Mohsenin, D. Truong, and B. Baas, *Multi-split-row threshold decoding implementations for LDPC codes*, International Symposium on Circuits and Systems (2009).

[N$^+$08] J. Nickolls et al., *Scalable parallel programming with CUDA*, ACM Queue **6** (2008), 40–53.

[NDB09] N. Nguyen, A. Dominguez, and R. Barua, *Memory allocation for embedded systems with a compile-time-unknown scratch-pad size*, ACM Trans. Embed. Comput. Syst. **8** (2009), 21:1–21:32.

[Nio] Nios, www.altera.com/nios.

[P$^+$95] J. L. Pino et al., *Software synthesis for DSP using Ptolemy*, Journal of VLSI Signal Processing Systems **9** (1995), no. 1-2, 7–21.

[P$^+$08] W. Plishker et al., *Functional DIF for rapid prototyping*, Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping, 2008, pp. 17–23.

[PBSV$^+$06] A. Pinto, A. Bonivento, A. L. Sangiovanni-Vincentelli, R. Passerone, and M. Sgroi, *System level design paradigms: Platform-based design and communication synthesis*, **11** (2006), no. 3, 537–563.

[PD10] J. Park and W. J. Dally, *Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures*, Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, SPAA '10, 2010, pp. 1–10.

[PTD$^+$06] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins, *Deterministic parallel processing*, Int. J. Parallel Program. **34** (2006), 323–341.

[Pul08] D. Pulley, *Multi-core dsp for base stations: large and small*, Proceedings of the 2008 Asia and South Pacific Design Automation Conference, ASP-DAC '08, 2008, pp. 389–391.

[Rau94] R. Rau, *Iterative modulo scheduling: an algorithm for software pipelining loops*, International Symposium on Microarchitecture (1994), 63–74.

[RGSL96] J. Ruttenbergand, G. Gao, A. Stoutchinin, and W. Lichtenstein, *Software pipelining showdown: optimal vs. heuristic methods in a production compiler*, Conference on Programming Language Design and Implementation (1996).

[RTM$^+$09] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Vora, *A 45nm 8-core enterprise xeon processor*, Solid-State Circuits Conference, 2009. A-SSCC 2009. IEEE Asian, nov. 2009, pp. 9 –12.

[Sar89] V. Sarkar, *Determining average program execution times and their variance*, Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, PLDI '89, 1989, pp. 298–312.

[SBGC07] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, *Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs*, Design Automation Conference (2007).

[SDS$^+$11] S. Sawant, U. Desai, G. Shamanna, L. Sharma, M. Ranade, A. Agarwal, S. Dakshinamurthy, and R. Narayanan, *A 32nm Westmere-EX Xeon enterprise processor*, Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International, feb. 2011, pp. 74 –75.

[SGB08] S. Stuijk, M. Geilen, and T. Basten, *Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs*, IEEE Transactions on Computers **57** (2008), no. 10, 1331–1345.

[SGM$^+$05] J. Sztipanovits, C. J. Glossner, T. N. Mudge, C. Rowen, A. L. Sangiovanni-Vincentelli, W. Wolf, and F. Zhao, *Panel session: Grand challenges in embedded systems*, International Conference on Embedded Software (2005), 333.

[SK03] R. Szymanek and K. Krzysztof, *Partial task assignment of task graphs under heterogeneous resource constraints*, Proceedings of the 40th annual Design Automation Conference (New York, NY, USA), DAC '03, ACM, 2003, pp. 244–249.

[SL05] H. Sutter and J. Larus, *Software and the concurrency revolution*, ACM Queue **3** (2005), no. 7, 54–62.

[SVCDBS04] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, *Benefits and challenges for platform-based design*, Design Automation Conference, 2004. Proceedings. 41st, 2004, pp. 409 – 414.

[TA10] W. Thies and S. Amarasinghe, *An empirical characterization of stream programs and its implications for language and compiler design*, Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, 2010, pp. 365–376.

[Tah03] W. Taha, *A gentle introduction to multi-stage programming*, Domain-Specific Program Generation, LNCS (2003), 30–50.

[TB10] D. N. Truong and B. M. Baas, *Massively parallel processor array for mid-/back-end ultrasound signal processing*, Biomedical Circuits and Systems Conference (BioCAS), 2010 IEEE, Nov. 2010, pp. 274–277.

[TCM$^+$08] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas, *A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling*, VLSI Circuits, 2008 IEEE Symposium on, june 2008, pp. 22 –23.

[TCM$^+$09] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, *A 167-processor computational platform in 65 nm CMOS*, Solid-State Circuits, IEEE Journal of **44** (2009), no. 4, 1130–1144.

[Thi07] L. Thiele, *Performance analysis of distributed embedded systems*, International Conference on Embedded Software (2007).

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe, *Streamit: A language for streaming applications*, Proceedings of the 11th International Conference on Compiler Construction, CC '02, 2002, pp. 179–196.

[TKS$^+$05] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe, *Teleport messaging for distributed stream programs*, Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, 2005, pp. 224–235.

[TLA03] W. Thies, J. Lin, and S. Amarasinghe, *Partitioning a structured stream graph using dynamic programming*, Tech. report, CS Department, Massachusetts Institute of Technology, 2003.

[UDB06] S. Udayakumaran, A. Dominguez, and R. Barua, *Dynamic allocation for scratch-pad memory using compile-time decisions*, ACM Trans. Embed. Comput. Syst. **5** (2006), 472–511.

[VR99] R. D. Venkataramana and N. Ranganathan, *A learning automata based framework for task assignment in heterogeneous computing systems*, Proceedings of the 1999 ACM symposium on Applied computing (New York, NY, USA), SAC '99, ACM, 1999, pp. 541–547.

[WBGB10] M. Wiggers, M. Bekooij, M. Geilen, and T. Basten, *Simultaneous budget and buffer size computation for throughput-constrained task graphs*, Proceedings of the Conference on Design Automation and Test in Europe (2010).

[WEE$^+$08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, *The worst-case execution-time problem: overview of methods and survey of tools*, ACM Trans. Embed. Comput. Syst. **7** (2008), 36:1–36:53.

[Wen75] K. S. Weng, *Stream oriented computation in recursive data flow schemas*, Tech. report, Massachusetts Institute of Technology, 1975.

[WKP11] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, *Fermi GF100 GPU architecture*, Micro, IEEE **31** (2011), no. 2, 50 –59.

[XB08] Z. Xiao and B. M. Baas, *A high-performance parallel cavlc encoder on a fine-grained many-core system*, International Conference on Computer Design, (ICCD '08), 2008, pp. 248–254.

[YMA+06] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. M. Baas, *An asynchronous array of simple processors for DSP applications*, IEEE International Solid-State Circuits Conference (2006).

[ZL06] Y. Zhou and E. A. Lee, *A causality interface for deadlock analysis in dataflow*, Proceedings of the 6th ACM and IEEE International conference on Embedded software, EMSOFT '06, 2006, pp. 44–52.

[ZTB00] E. Zitzler, J. Teich, and S. S. Bhattacharyya, *Multidimensional exploration of software implementationsfor dsp algorithms*, J. VLSI Signal Process. Syst. **24** (2000), 83–98.

APPENDIX A

Benchmark Applications

This appendix presents the benchmark applications in our new malleable dataflow model. Details are presented in Chapter 6. The applications are parallel merge sort (SORT), matrix multiplication (MMUL), fast fourier transform (FFT), low-density parity check (LDPC) and advanced encryption standard (AES). Figures A.1 to A.5 present graphical representations of the applications.

The task specification starts with a list of forming parameters and their type. The `interface` section specifies the set of input and output ports of the task, and the `function` section specifies its data transformation function, all based on the given parameters.

Similarly, application specification also starts with a list of forming parameters. The `interface` section is the same as task interface. In a `composition` section, the tasks are instantiated by assigning the corresponding parameters using the `instantiate` construct. The channels are instantiated using the `connect` construct which connects ports of two tasks.

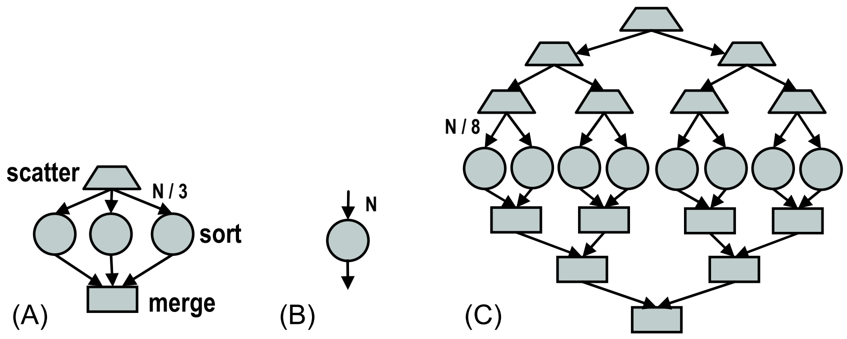


FIGURE A.1. Parallel merge sort application constructed with A) $\Phi = (3, 3)$, B) $\Phi = (1, 1)$ and C) $\Phi = (8, 2)$.

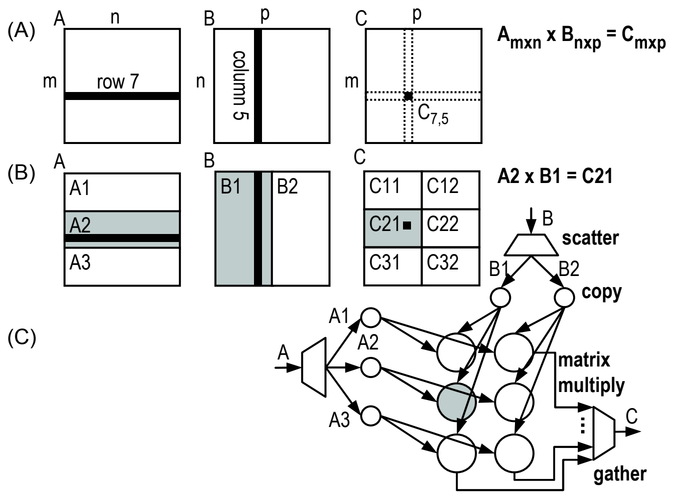


FIGURE A.2. A) Matrix multiply. B) Parallelized matrix multiply for $\Phi = (3,2)$. C) Task graph with $\Phi = (3,2)$.

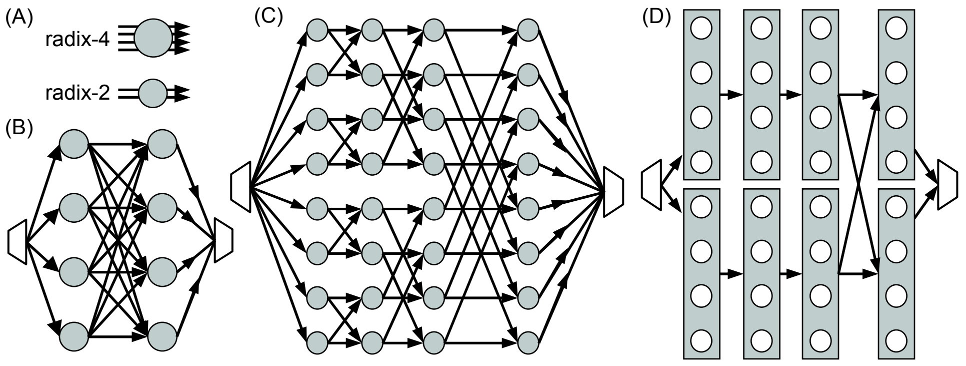


FIGURE A.3. A) Radix-2 and radix-4 butterfly tasks. B) 16-point FFT application with radix-4 butterfly tasks. C) The same FFT computed with radix-2. D) Radix-2 task graph condensed with a factor of 4.

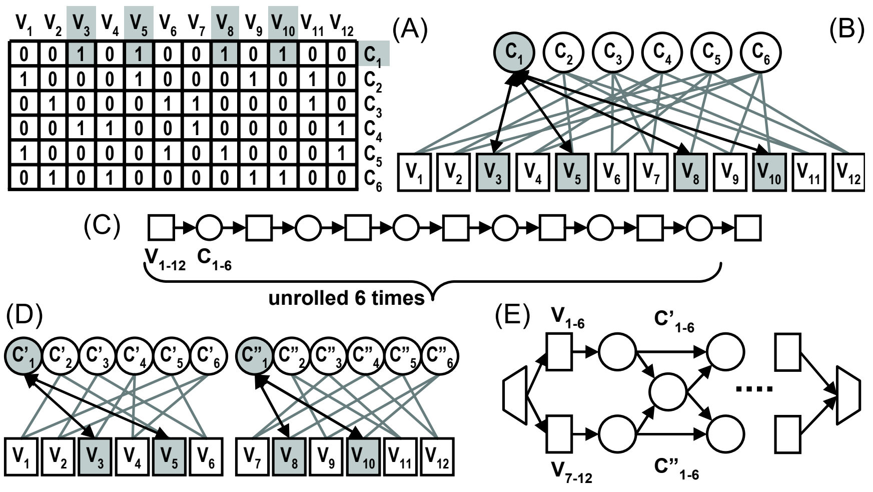


FIGURE A.4. LDPC application: A) Sample H matrix. B) Tanner graph. C) Task graph. Row-Split LDPC with split factor $\phi = 2$: D) Tanner graph. E) Task graph.

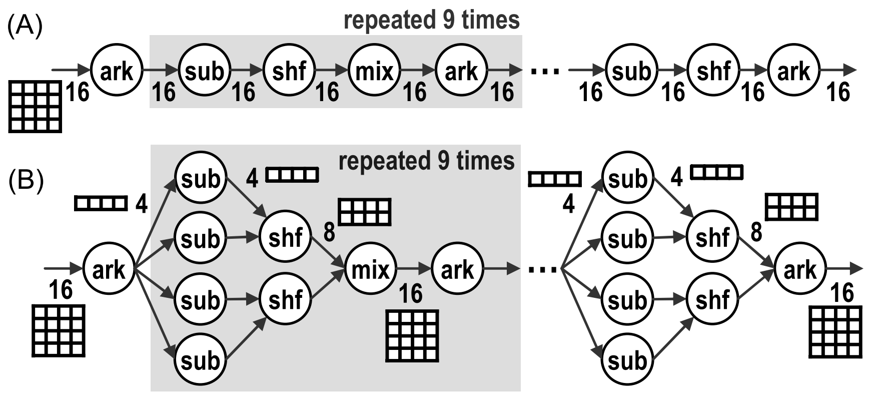


FIGURE A.5. AES application: A) $\Phi = (1, 1, 1, 1)$ B) $\Phi = (4, 2, 1, 1)$.

```
///////////////////////////////////////
///////////////////////////////////////
////                               ////
////             SORT              ////
////                               ////
///////////////////////////////////////
///////////////////////////////////////

//scatter input array into K sub-arrays:
actor Scatter ( int N, //length of the input array
                int K  //number of the output arrays (out-degree)
              ){
  interface {
    input input_array ( N );
    for ( i=0; i < K; i++ )
      output sub_array[i] ( N/K );
  }
  function {
    //scatter data
  }
}

//sort an array:
actor QuickSort ( int N //length of input and output arrays
                ){
  interface {
    input  unsorted_array ( N );
    output sorted_array ( N );
  }
  function {
    //the quicksort algorithm
  }
}

//merge input sorted arrays into one larger output array:
actor Merge ( int   N, //length of the output array
              int   K  //number of the input arrays (in-degree)
            ){
  interface {
    output merged_array ( N );
    for ( i=0; i < K; i++ )
      input sub_array[i] ( N/K );
  }
  function {
    //merge based on the method in mergesort algorithm
  }
}

//merge input sorted arrays into one larger output array
//using a tree network constructed out of merge actors:
composite_actor MergeNetwork ( int N, //length of the output array
                               int K, //in-degree of merge actors
                               int D  //depth of the tree
                             ){
  interface {
    output merged_array ( N );
    for (i=0; i < K^D; i++)
      input sub_array[i] ( N / K^D );
  }
  composition {
    //instantiate merge actors
    for (d=D-1; d >= 0; d--)
    for (i=0; i < K^d; i++)
      instantiate Merge merge[d][i] ( N / K^d, K, null );

    //connections
    for (i=0; i < K^D; i++)
      connect ( sub_array[i], merge[D-1][i/K].sub_array[i%K] );

    for (d=D-1; d > 0; d--)
    for (i=0; i < K^d; i++)
      connect ( merge[d][i].merged_array, merge[d-1][i/K].sub_array[i%K] );

    connect ( merge[0][0].merged_array, merged_array );
  }
}

//scatter the input array using a tree network of scatter actors:
composite_actor ScatterNetwork ( int N, //length of the input array
                                 int K, //out-degree of scatter actors
                                 int D  //depth of the tree
                               ){
//similar to MergeNetwork...
}

//sort an array using a ScatterNetwork to distribute the input
//array among a number of QuickSort actors, and a MergeNetwork
//to merge the sorted sub-arrays into the final output array:
composite_actor MergeSort ( int N, //length of input and output arrays
                            int K, //in-degree of merge actors in the tree
                            int D  //depth of the tree
                          ){
  interface {
    input input_array ( N );
    output output_array ( N );
  }
  composition {

    if (D==0) {
      instantiate QuickSort quicksort ( N );
      connect ( input_array,            quicksort.unsorted_array );
      connect ( quicksort.sorted_array, output_array             );
    } else {

      //instantiate a scatter network which distributes
      //the input array among K^D sort actors
      instantiate ScatterNetwork scatternet ( N, K, D );

      //instantiate the sort actors
      for (i=0; i < K^D; i++)
        instantiate QuickSort quicksort[i] ( N / K^D );

      //instantiate a merge network which merges
      //the sorted sub-arrays into the output array
      instantiate MergeNetwork mergenet ( N, K, D );

      //connections:
      connect ( input_array, scatternet.input_array );

      for (i=0; i < K^D; i++) {
        connect ( scatternet.sub_array[i],    quicksort[i].unsorted_array );
        connect ( quicksort[i].sorted_array,  mergenet.sub_array[i]       );
      }

      connect ( mergenet.merged_array, output_array );
    }
  }
}
```

```
////////////////////////////////////////
////////////////////////////////////////
////                                ////
////             FFT                ////
////                                ////
////////////////////////////////////////
////////////////////////////////////////

actor Butterfly ( int i,j, //position of this butterfly group
                  int K,   //radix
                  int K2D  //# of butterflies
                ){
  interface {
    input  in ( K*K2D );
    output out ( K*K2D );
  }
  function {
    //K2D butterflies of radix K in position [i][j]
  }
}

composite_actor FFT ( int N,  //size of input/output array
                      int K,  //degree of butterfly tasks
                      int D   //condense K2D=K^D butterflies into one
                     ){
  interface {
    input input_array ( N );
    output output_array ( N );
  }
  composition {

    int numC = (int)( Math.log(N)/Math.log(K) ); //# of columns
    int numR = N/K;                              //# of rows
    int K2D  = powi(K,D);
    int i,j,x,X,b,B,j1,j2,jj1,jj2;

    //butterfly tasks
    for (i=0; i<numC;     i++)
    for (j=0; j<numR/K2D; j++)
      instantiate Butterfly bf[i][j] (i,j, K, K2D);

    //connections
    for (i=0; i<D;       i++)
    for (j=0; j<numR/K2D; j++)
      connect( bf[i][j], bf[i+1][j] );

    //butterfly connections
    for (i=D, B=(numR/K2D)/K, X=1; i<numC-1; i++, B/=K, X*=K)
      for (b=0; b<B; b++)
      for (x=0; x<X; x++)
        for (jj1=0, j1=b*X*K+x; jj1<K; jj1++, j1+=X)
        for (jj2=0, j2=b*X*K+x; jj2<K; jj2++, j2+=X)
          connect( bf[i][j1], bf[i+1][j2] );

    if (numR>K2D) {
      ScatterNetwork sc (N,numR/K2D);
      GatherNetwork ga (N,numR/K2D);
      //connections
      for (j=0; j<numR/K2D; j++) {
        connect( sc.out[j], bf[0][j].in );
        connect( bf[numC-1][j], ga.in[j] );
      }
      connect( input_array, sc.in );
      connect( ga.out, output_array );
    }
    else {
      connect( input_array, bf[0][0].in );
      connect( bf[numC-1][0].out, output_array );
    }
  }
```

```
///////////////////////////////////////
///////////////////////////////////////
////                               ////
////             LDPC              ////
////                               ////
///////////////////////////////////////
///////////////////////////////////////

composite_actor LDPC ( int c //row-split factor
                     ){
  interface {
    input input_array ( 2048 );
    output output_array ( 2048 );
  }
  composition {

    int logc = (int)(Math.log(c)/Math.log(2));
    int c2id=c, nlid,r1id,gaid,r2id,scid,r3id,r4id;

    //Zero Alpha
    instantiate Scatter scat (2048,c);
    connect (input_array_, scat.in);
    for (j=0; j<c; j++)
      connect( scat.out[j], ldpctask[c2id+j]);

    //LDPC network:
    for (int irep=0; irep<rep; irep++) {

      nlid = c + c2id;
      r1id = c + nlid;
      gaid = c + r1id;
      r2id = numNetwork(2,logc) + gaid;
      scid = 1 + r2id;
      r3id = numNetwork(2,logc) + scid;
      r4id = r1id;

      //C2 Task
      for (j=0; j<c; j++) {
        instantiate LDPC_C2 ldpctask[c2id+j] (c,j);
        connect( ldpctask[c2id+j], ldpctask[r1id+j] );
      }

      //id for next level C2
      if (c==1) c2id=r4id+c; else c2id=r3id+c;

      //R Tasks:
      if (c==1) {
        //R4
        instantiate LDPC_R4 ldpctask[r4id] (c);
        connect( ldpctask[r4id], ldpctask[c2id] );
      }else{//c>1

        //gather
        instantiate GatherNetwork ga (384*2*c,logc);

        //scatter
        instantiate ScatterNetwork sc (384*2*c,logc);

        //R1
        for (j=0; j<c; j++) {
          instantiate LDPC_R1 ldpctask[r1id+j] (c);
          connect( ldpctask[r1id+j], ga.in[j] );//to gather
          connect( ldpctask[r1id+j], ldpctask[r3id+j] );//to R3
        }

        //R2
        connect( ldpctask[r2id-1], ldpctask[r2id] );//from gather
        instantiate LDPC_R2 ldpctask[r2id] (c);
        connect( ldpctask[r2id], ldpctask[r2id+1] );
        //R3
        for (j=0; j<c; j++) {
          connect( sc.out[j], ldpctask[r3id+j] );//from scatter
          instantiate LDPC_R3 ldpctask[r3id+j] (c,j);
          connect( ldpctask[r3id+j], ldpctask[c2id+j] );//to C2 next level
        }
      }//if c>1
    }//for rep

    int gathid = c2id + c + numG(c) - 1;
    //C2
    for (j=0; j<c; j++) {
      instantiate LDPC_C2 ldpctask[c2id+j] (c,j);
      if (c>1) //bit -> gather
        connect( ldpctask[c2id+j], ldpctask[gathid - numG(c) + 1 + j/4] );
    }

    //Gather
    if ((c==2)||(c==4)) {
      instantiate Gather ldpctask[gathid] (2048,c);
      connect( ldpctask[gathid].out, output_array );
    }
    else if ((c==8)||(c==16)) {
      for (j=0; j<c/4; j++) {
        int gid = gathid - numG(c) + 1 +j;
        instantiate Gather ldpctask[gid] (4*2048/c,4);
        connect( ldpctask[gid], ldpctask[gathid] );
      }
      instantiate Gather ldpctask[gathid] (2048,2);
    }
  }
}
```

```
///////////////////////////////////////
///////////////////////////////////////
////                               ////
////             AES               ////
////                               ////
///////////////////////////////////////
///////////////////////////////////////

actor SubB ( int p, int inputs, int outputs) {
  interface {
    for ( i=0; i < inputs; i++ ) input in[i] ( 16/(p*inputs) );
    for ( i=0; i < outputs; i++ ) output out[i] ( 16/(p*outputs) );
  }
  function {
    //substitute byte algorithm in AES application
  }
}

actor AddRK ( int p, int roundNr, int pos, int inputs, int outputs) {
  interface {
    for ( i=0; i < inputs; i++ ) input in[i] ( 16/(p*inputs) );
    for ( i=0; i < outputs; i++ ) output out[i] ( 16/(p*outputs) );
  }
  function {
    //add round key algorithm in AES application
  }
}

actor ShiftR ( int p, int pos, int inputs, int outputs) {
  interface {
    for ( i=0; i < inputs; i++ ) input in[i] ( 16/(p*inputs) );
    for ( i=0; i < outputs; i++ ) output out[i] ( 16/(p*outputs) );
  }
  function {
    //shift row algorithm in AES application
  }
}

actor MixC ( int p, int pos, int inputs, int outputs) {
  interface {
    for ( i=0; i < inputs; i++ ) input in[i] ( 16/(p*inputs) );
    for ( i=0; i < outputs; i++ ) output out[i] ( 16/(p*outputs) );
  }
  function {
    //mix column algorithm in AES application
  }
}

composite_actor AES ( int sub,   //# of SubB tasks
                      int shift, //# of ShiftR tasks
                      int addRK, //# of AddRK tasks
                      int mixc   //# if MixC tasks
                     ){
  interface {
    input input_array ( 16 );
    output output_array ( 16 );
  }
  composition {

    int i = 1, hold = 1, shiftPos, subPos, addRKPos, inputs, outputs;
    addRKPos = i + 1;
    if (addRK > 1) {
      instantiate Scatter aestask[i] (16, addRK);
      i++;
      for (int k = 0; k < addRK; k++)
        connect( aestask[i-1].out[k], aestask[addRKPos++].in[0] );
      hold++;
    }

    //insert AddRK tasks
```

```
subPos = hold + addRK;
for (int h = 0; h < addRK; h++) {
  if (sub <= addRK)
    outputs = 1;
  else
    outputs = sub / addRK;
  instantiate AddRK aestask[i] (addRK, 0, h * (16/addRK), 1, outputs);
  i++;
  if (sub > addRK)
    for (int k = 0; k < (sub / addRK); k++)
      connect(aestask[i-1], aestask[subPos++]);
}

//Unrolled 9 times:
for (int j = 0; j < 9; j++) {
  hold = i - addRK;
  shiftPos = i + sub;

  //insert SubB tasks
  for (int h = 0; h < sub; h++) {
    if (sub <= addRK)
      for (int k = 0; k < addRK / sub; k++)
        connect(aestask[hold++], aestask[i]);

    if (addRK <= sub)
      inputs = 1;
    else
      inputs = addRK / sub;
    if (shift <= sub)
      outputs = 1;
    else
      outputs = shift / sub;
    instantiate SubB aestask[i] (sub, inputs, outputs);
    i++;
    if (sub < shift)
      for (int k = 0; k < (shift / sub); k++)
        connect(aestask[i-1], aestask[shiftPos++]);
  }

  //insert ShirtR tasks
  hold = i - sub;
  for (int h = 0; h < shift; h++) {
    if (sub >= shift)
      for (int k = 0; k < sub / shift; k++)
        g.addEdge(hold++, i, 16 / sub, ITYPE,STYPE);
    if (sub <= shift)
      inputs = 1;
    else
      inputs = sub / shift;
    instantiate ShiftR aestask[i] (shift, h*(4/shift), inputs, 1);
    i++;
    connect(aestask[i-1], aestask[i + shift - h - 1]);
  }

  if (shift > 1) {
    instantiate Gather aestask[i] (16, shift);
    i++;
    connect(aestask[i-1], aestask[i]);
  }

  //insert MixC task
  instantiate MixC aestask[i] (1);
  i++;

  if (addRK > 1) {
    connect(aestask[i-1], aestask[i]);
    instantiate Scatter aestask[i] (addRK);
    i++;
  }
```

```
    //insert AddRK tasks
    hold = i - 1;
    subPos = i + addRK;
    for (int h = 0; h < addRK; h++) {
      connect(aestask[hold], aestask[i]);
      if (sub <= addRK)
        outputs = 1;
      else
        outputs = sub / addRK;
      instantiate AddRK aestask[i] (addRK, j + 1,  h * (16/addRK), 1, outputs);
      i++;
      if (addRK< sub)
        for (int k = 0; k < (sub / addRK); k++)
          connect (aestask[i-1], aestask[subPos++]);
    }
  }

  hold = i - addRK;
  shiftPos = i + sub;

  //insert SubB tasks
  for (int h = 0; h < sub; h++) {
    if (addRK>= sub)
      for (int k = 0; k < addRK / sub; k++)
        connect (aestask[hold++], aestask[i]);
    if (addRK <= sub)
      inputs = 1;
    else
      inputs = addRK / sub;
    if (shift <= sub)
      outputs = 1;
    else
      outputs = shift / sub;
    instantiate SubB aestask[i] (sub, inputs, outputs);
    i++;
    if (sub< shift)
      for (int k = 0; k < (shift / sub); k++)
        connect (aestask[i-1], aestask[shiftPos++]);
  }

  //insert ShiftR tasks
  hold = i - sub;
  addRKPos = i + shift;
  for (int h = 0; h < shift; h++) {
    if (sub >= shift)
      for (int k = 0; k < sub / shift; k++)
        connect (aestask[hold++], aestask[i]);
    if (sub <= shift)
      inputs = 1;
    else
      inputs = sub / shift;
    if (addRK <= shift)
      outputs = 1;
    else
      outputs = addRK / shift;
    instantiate ShiftR aestask[i] (shift, h*(4/shift), inputs, outputs);
    i++;
    if (shift< addRK)
      for (int k = 0; k < (addRK / shift); k++)
        connect (aestask[i-1], aestask[addRKPos++]);
  }

  //insert AddRK tasks
  hold = i - shift;
  for (int h = 0; h < addRK; h++) {
    if (shift>= addRK)
      for (int k = 0; k < (shift / addRK); k++)
        connect (aestask[hold++], aestask[i]);
    if (shift <= addRK)
      inputs = 1;
      else
        inputs = shift / addRK;
      instantiate AddRK aestask[i] (addRK, 10,  h * (16/addRK), inputs, 1);
      i++;
    }
    if (addRK > 1) {
      hold = i - addRK;
      for (int k = 0; k < addRK; k++) {
        connect( aestask[hold++], aestask[i]);
      }
      instantiate Gather aestask[i] (addRK);
      i++;
    }
  }
}
```