Compiler Parallelization Techniques for Tiled Multicore Processors

Donghwan Jeon
Department of Computer Science and Engineering
University of California San Diego
9500 Gilman Drive, La Jolla, CA, USA 92093
djeon@cs.ucsd.edu

ABSTRACT
Recently, tiled multicore processors have been proposed as a solution to provide both performance and scalability. Unlike conventional multicore processors, tiled microprocessors provide on-chip networks to exploit fine-grained parallelism. However, the performance of tiled microprocessors largely depends on compilers because of their relatively simple hardware; exploitation of parallelism, communication between cores, memory locality, and load balancing are all managed by a compiler. Thus, the compiler design is very important in tiled multicore processors. In this paper, we explore previously proposed compiler parallelization techniques for tiled processors, focusing on the exploitation of fine-grained parallelization. We also investigate future research directions in this area.

1. INTRODUCTION
For decades, microprocessor performance has improved at an exponential rate. By exploiting ever increasing transistor budgets, computer architects have increased processor frequency and instructions per cycle (IPC), which collectively have enabled the dramatic improvements. However, the microprocessor industry is now undergoing a fundamental change. Although Moore's law is still valid, conventional processor design techniques are bringing diminishing returns. Techniques to increase the operating frequency, such as deeper pipelining, suffer from prohibitive power consumption. Furthermore, techniques to increase IPC, including wider-issue design and complicated branch prediction, do not bring any substantial speedup due to their negative impacts on the processor frequency [1]. Industry and academia are seeking alternative approaches to processor design that will allow for continued performance gains.

As an alternative to conventional processors, chip multiprocessors (CMPs) [15] are receiving much attention. Unlike conventional monolithic microprocessors, a CMP consists of several simple cores on a chip. With multiple on-chip cores, CMPs provide new opportunities to exploit thread-level parallelism. To take advantage of thread-level parallelism, several parallelization techniques were proposed to convert loop-level parallelism into thread-level parallelism on CMPs [23]. These techniques concurrently execute multiple loop iterations across multiple cores. The reduced latency of on-chip communication through shared memory helps efficient synchronization and data transfer between threads. Unfortunately, parallelization on CMPs has remained largely restricted to coarse-grained parallel loops because inter-thread communication costs are still too expensive to exploit fine-grained ILP.

Tiled processors [19] were proposed to enable the exploitation of fine-grained parallelism that is lacking in CMPs. Tiled processors are differentiated from CMPs by inclusion of a low-cost on-chip network called Scalar Operand Network (SON). The SON is designed to efficiently transport scalar values across cores, enabling the exploitation of fine-grained as well as coarse-grained parallelism.

To enable scalability, tiled processors delegate to the compiler many tasks that were previously performed in hardware. As a result, the main challenge of tiled processors lies in the design of the compiler. For example, the compiler in a tiled processor decides where an instruction is executed whereas it is determined by hardware issue logic at run-time in superscalar processors. Among all the tasks a parallelizing compiler performs in tiled processors, the two most important tasks are to find both coarse-grained and fine-grained parallelism in a program and to convert it into thread-level parallelism. The combination of these two requirements is unique to compilers for tiled processors. Compilers for single-core processors enhance fine-grained parallelism but they do not convert it into thread-level parallelism. Conversely, compilers for conventional parallel systems do not exploit fine-grained parallelism.

In this paper we survey compiler parallelization techniques for tiled multicore processors. Specifically, we investigate how they detect parallelism and how they convert the detected parallelism into thread-level parallelism. Our survey is focused on fine-grained parallelization rather than coarse-grained parallelization. Note that coarse-grained parallelization techniques for a CMP can also be used in tiled processors. Additionally, we focus on automatic parallelization rather than explicit parallelization by programmers. Explicit parallel programming can be helpful in finding parallelism but it might decrease the productivity of programmers by placing more of a burden on them. Moreover, automatic compiler parallelization is the only feasible solution for dealing with the large body of legacy sequential programs.

This paper is organized as follows. Section 2 introduces general background on compiler parallelization, specifically dependences and parallelism from a compiler writer's view. Section 3 reviews conventional coarse grained parallelization techniques. Tiled multicore processors and their implica-

tions for compilers are discussed in Section 4. Section 5 then shows three parallelization techniques for tiled processors in depth and discusses their advantages and disadvantages. We present possible directions for future research in Section 6 before concluding the paper in Section 7.

2. PARALLELIZATION BACKGROUND

2.1 Dependence

A parallelization technique is a code transformation process that exposes thread-level parallelism. A *valid* code transformation is one that does not change the execution results of the original program, and a parallelization technique must be *valid* unless it is a speculative one. But how can we guarantee the validity of a code transformation? According to Horwitz et al [10] in their Equivalence Theorem, two programs with the same dependence relations are equivalent. In other words, any *valid* code transformation, including parallelization techniques, must respect all the dependences in the original program.

There are two kinds of dependences: control dependence and data dependence. Their formal definitions are as follows [11]:

- **Control Dependence** A statement y is said to be control dependent on another statement x if (1) there exists a nontrivial path from x to y such that every statement z != x in the path is postdominated by y, and (2) x is not postdominated by y.
- **Data Dependence** There is a data dependence from statement S1 to statement S2 (statement S2 depends on statement S1) if and only if (1) both statements access the same memory location and at least one of them stores into it, and (2) there is a feasible run-time execution path from S1 to S2.

Data dependences can be categorized in a few ways. First, there are three kinds of data dependences depending on the type of operations: flow dependence (write to read), anti dependence (read to write), and output dependence (write to write). Data dependences can be also viewed based on the type of storage (register or memory object) or their relation to loop iterations (loop-carried or loop-independent). Unlike loop-independent dependences, a loop-carried dependence is a data dependence between statements in different loop iterations.

Control dependences can be computed by building a control-flow graph (CFG) and a post-dominance tree [6]. The detection of data dependences depends on the type of dependence. Register data dependences can be easily and precisely computed by data-flow analysis. In fact, the use of the SSA form, a widely used technique in compilers, even eliminates the necessity of data-flow analysis. Anti- and output- register data dependences are removed by using the SSA form, and data dependences can be detected by simply comparing variable names [5]. The challenging part lies in memory dependence analysis. Since memory address is calculated at run-time, a compiler has only a limited amount of information to statically determine precise memory aliases, leading to very conservative results. Clearly, the more conservative a

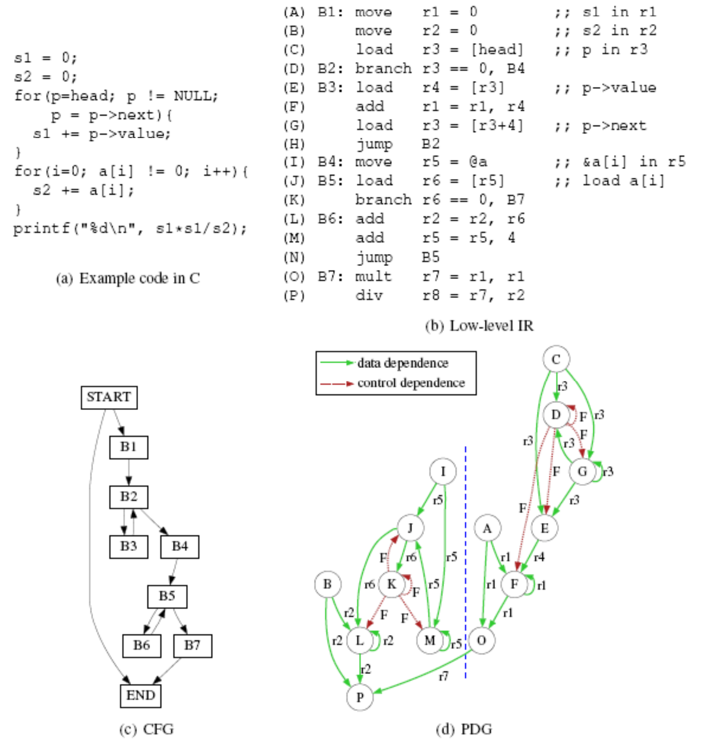


Figure 1: (a) Example code in C (b) Low-level IR, (c) CFG, (d) PDG[16]

dependence analyzer is, the harder it becomes to apply parallelization techniques. Indeed, this is the primary obstacle in many parallelizing compilers.

Another concern is how to represent and manage detected dependences in the compiler. Among many proposed representations to describe dependences in a program, PDG [7] (program dependence graph) is especially useful since it contains both control and data dependences, allowing compilers to check available dependences in a single pass. Figure 1 shows the PDG building process from a C source code: first a CFG is built from low-level intermediate representation (IR), then a post-dominance tree is created to build control dependence information, and finally a PDG is built after data dependence analysis. In Figure1 (d), data dependences are represented by solid arcs, while control dependences are represented by dotted arcs.

2.2 Parallelism Classification

To perform parallelization, a compiler has to discover parallelism in the input program. Parallelism can be classified in two ways: based on granularity and scope.

Based on the granularity of work that can be independently performed, parallelism can be coarse-grained or fine-grained. The unit of typical coarse-grained parallelism is a procedure or a loop iteration, while that of fine-grained parallelism is typically several instructions. Coarse-grained parallelism can be exploited in most multiprocessor systems, since each work unit could benefit from enough parallelism even after paying communication cost for dependence resolutions. On contrast, fine-grained parallelism usually requires spe-

cial hardware support to satisfy dependences at a low cost. Although fine-grained parallelism is hard to exploit without hardware support, it has an advantage in parallelism detection since fine-grained parallelism such as ILP can be easily found in most code regions of a program. Coarse-grained parallelism is evident in many numerical or scientific programs, but it is rarely found in general purpose programs.

Alternatively, parallelism can be characterized by its scope. A compiler can extract parallelism in a basic block, in a loop nest, or in an entire procedure. Specifically, the basic block parallelism extraction is restricted to available parallelism only within a basic block, while loop boundary parallelism extraction allows instructions in different iterations to be simultaneously executed; procedure boundary parallelism extraction even allows the concurrent execution of two independent code regions such as two independent loops. As the scope gets larger, potentially more parallelism could be extracted at the expense of the increased compiler complexity. The notable problems introduced by extended parallelism scopes are the handling of control-flows and load balancing. In Section 5, we will examine how these issues are treated.

3. COARSE-GRAINED PARALLELIZATION

In this section, we will briefly review two coarse-grained parallelization techniques: DOALL and DOACROSS. These parallelization techniques can be used in tiled multicore processors as well as in other conventional multi-processors.

3.1 DOALL Parallelization

A DOALL loop is a loop where every iteration is independent from other iterations. In other words, DOALL loops do not have any loop-carried dependence. DOALL loop parallelization has been very popular for several reasons. First, loops account for a significant portion of the execution time in many applications. According to Amdahl's law, it is better to focus on parallelizing code regions where most of the execution time is spent. Second, DOALL loops are efficiently parallelizable. Because a DOALL loop does not have any loop-carried dependence, no data communication is needed to satisfy dependences after parallelization. Third, good load balancing can be easily achieved in DOALL parallelization since each iteration has the same number of instructions. Finally, a scalable parallelization is possible provided that the trip count of a loop is large enough. An example of DOALL parallelization is illustrated in Figure 2.

DOALL loops have been extensively studied in two directions: how to prove a loop is DOALL and how to transform a non-DOALL loop into a DOALL or improve the efficiency of DOALL execution. The challenging part in the detection of DOALL is proving loop iterations are free from memory dependences. Although many dependence test techniques have been developed, including GCD test and Omega test [11], they are still conservative and therefore miss many parallelization opportunities. To overcome conservative memory analyses, speculative DOALL parallelizations were proposed [23]. These speculative techniques are beyond the scope of this paper.

Many loop transformation techniques have been developed to convert a non-DOALL loop to a DOALL loop. Scalar and array privatization techniques break output data dependences by using extra storage. Loop distribution splits a loop into multiple loops so that some of the resulting loops can be executed as DOALL. Loop skewing exposes hidden DOALL loops by manipulating a loop nest. Additionally, transformations making DOALL more efficient include loop fusion and loop interchange. These techniques make a loop body larger, increasing the effectiveness of the DOALL parallelization. These techniques are extensively used in High Performance Fortran (HPF) and SUIF [9] compilers and have been used to parallelize many scientific and numerical applications.

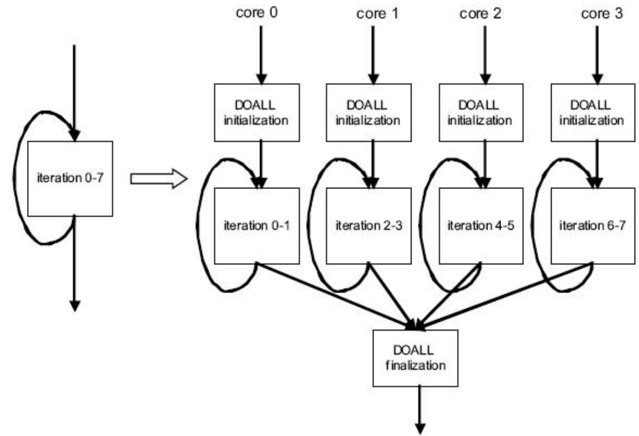


Figure 2: An example of DOALL parallelization[23]

Unfortunately, DOALL parallelization has been found to be very ineffective outside the scientific and numerical application domains. The main reason for this ineffectiveness is that arbitrary control-flows and irregular memory accesses via pointers force dependence analysis to be very conservative, leaving many DOALL parallelization opportunities undetected. Moreover, loops in such applications tend to have smaller loop bodies and lower iteration counts, allowing only moderate speedup by the DOALL parallelization.

3.2 DOACROSS Parallelization

DOACROSS [4] is a coarse-grained parallelization technique developed for loops with loop-carried dependences. In DOACROSS parallelization, loop iterations are executed on multiple processors in a round-robin fashion. The basic idea is that multiple loop iterations can simultaneously be executed until the point where a loop-carried dependence has to be satisfied. Although DOACROSS allows partially concurrent execution of multiple loop iterations, it is rarely used in practice for two reasons. First, the existence of loop-carried dependences limits the amount of parallelism. For each and every loop-carried dependence pair, a synchronization point is made and exploitable parallelism is restricted. Figure 5 shows the DOACROSS parallelization applied to a simple loop. As illustrated in the figure, loop-carried dependences cause couplings between loop iterations. Unfortunately, the cost of communication to satisfy a loop-carried dependence is too expensive in most parallel systems, which easily negates the benefit of parallelization. We will see how tiled processors alleviate this problem in the next section.

4. TILED MULTICORE PROCESSORS

4.1 Overview

Recently, several tiled multicore processors have been designed and proposed [21, 18]. The major goal of these tiled processors is to execute programs in a parallel fashion while providing scalability. To achieve scalability, a tiled processor consists of multiple simple cores where each core contains one or more functional units. Figure 3 shows a typical tiled processor. In tiled processors, multiple threads collectively execute the same program. Existing inter-thread dependences are quickly satisfied by on-chip networks designed to transport a scalar value across cores. Since the low-cost networks efficiently resolve inter-thread dependences, tiled processors can exploit not only coarse-grained parallelism but also fine-grained parallelism. Indeed, what differentiates tiled processors from other chip multiprocessors is the existence of the on-chip operand networks belonging to the class of scalar operand networks [20].

Another salient characteristic of tiled processors is that the orchestration of program execution is managed by a compiler. In most tiled processors, instruction assignment is statically performed by a compiler whereas superscalars dynamically assign the functional unit for each instruction at run-time. Although dynamic instruction assignment can increase the utilization of functional units, it significantly increases the complexity of hardware, prohibiting scalable implementations. The compiler-managed instruction scheduling is similar to that of a VLIW processor except that tiled processor compilers have more responsibility in communication management, memory locality management and control-flow management. We will discuss more details in Section 5.

4.2 Performance of Scalar Operand Networks

The low-cost on-chip networks in tiled processors are called scalar operand network (SON) because they are specialized for transporting a scalar value across cores. As the cost of communication gets cheaper, a processor can exploit more parallelization opportunities. For the metric of the SON performance, 5-tuple [20] was proposed, which characterizes the cost of fundamental functions of a SON. The 5-tuple <SO, SL, NHL, RL, RO> consists of the following cycle numbers:

- **Send occupancy:** average number of cycles that an ALU wastes in transmitting an operand to dependent instructions at other ALUs
- **Send latency:** average number of cycles incurred by the message at the send side of the network without consuming ALU cycles
- **Network hop latency:** average transport network hop latency, in cycles, between physically adjacent ALUs
- **Receive latency:** average number of cycles between when the final input to a consuming instruction arrives and when that instruction is issued
- **Receive occupancy:** average number of cycles that an ALU wastes by using a remote value.

The design of SON has a tradeoff between scalability and performance. For example, superscalar bypass networks achieve an ideal 5-tuple of <0, 0, 0, 0, 0> but they have only limited scalability. In contrast, the 5-tuple of typical scalable multiprocessor systems adds up to tens of or hundreds of cycles.

4.3 Case Study: MIT RAW

RAW [21] was the first designed tiled multicore processor. As shown in Figure 3, it contains 16 identical tiles connected by on-chip networks. Each RAW tile contains not only functional units but also a fetch unit, I-cache, and D-cache so that each tile is autonomous. In addition, two types of routers, one static and one dynamic, are contained in a tile so that they can handle the transportation of data. To deal with arbitrary communication patterns, the static router is configured using VLIW-style routing instructions stored in a separate router instruction memory. Both instruction code and switch code are generated by a compiler. The dynamic router in RAW is a typical wormhole router.

The designers of the RAW processor tried to minimize the 5-tuple of its SON in many ways. First, RAW provides a register-mapped data FIFO, eliminating send and receive occupancies. Furthermore, it employs a special bypass system so that data is sent to the router as soon as it is created, allowing a partial 5-tuple of <0, 0, 1, 1, 0> for routing - one cycle for inter-tile transportation and another cycle for router-to-compute processor transportation. Finally adding one cycle of wake-up receive latency, RAW static SON achieves a 5-tuple of <0, 0, 1, 2, 0>. These optimizations are shown in Figure 3. The result is that a compiler can exploit parallelism if the benefit of parallelization outweighs the 3-cycle data communication cost. This enables aggressive fine-grained parallelization.

5. FINE-GRAINED PARALLELIZATION

In this section, we first enumerate opportunities and challenges in parallelizing compilers for tiled multicore processors. We then examine three compiler techniques designed for tiled multicore processors. Unlike previous coarse-grained parallelization techniques, these techniques exploit the low-cost SON to respect dependences, performing fine-grained parallelization.

5.1 Compiler Opportunities and Challenges

Since the parallelization process largely determines the program execution time on tiled multicore processors, its importance cannot be overstated. As mentioned in previous sections, compilers have more responsibilities in tiled multicore processors, complicating the parallelization process more than ever. At the same time, tiled processors provide new opportunities not available in other processors. In this subsection, we will examine the opportunities and challenges of compilers for tiled processors. In the subsequent sections, we will see how these opportunities and challenges interact in existing compiler parallelization techniques.

5.1.1 Opportunities

Tiled processors offer new opportunities in the form of instruction level parallelism (ILP), memory parallelism, and multiple control-flow. However, it is up to the compiler

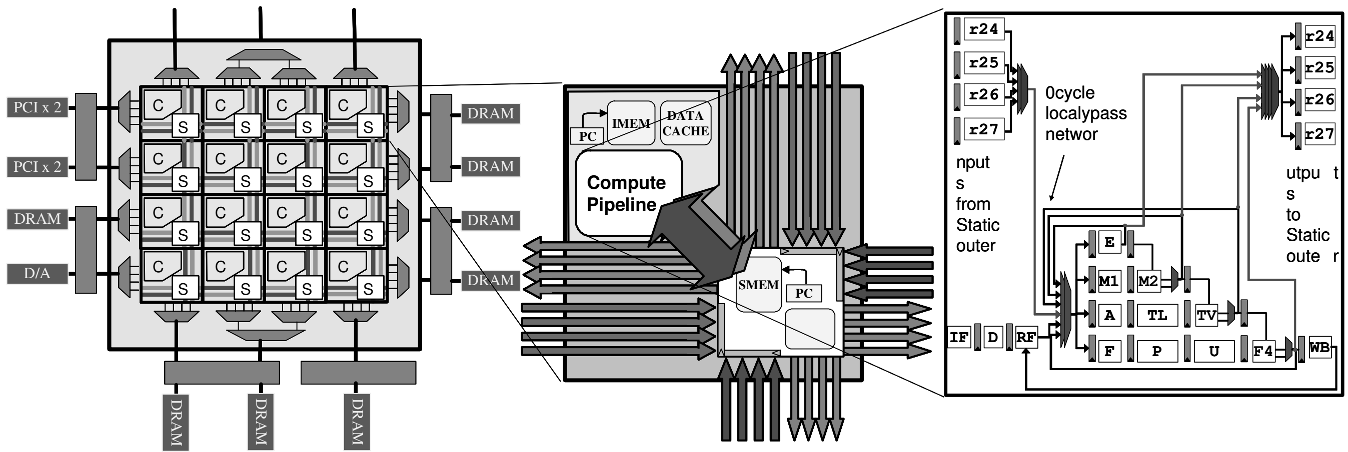


Figure 3: The Raw microprocessor consisting of 16 tiles[21]

to exploit these new opportunities. By effectively utilizing these opportunities, a parallelization compiler can significantly improve the performance of a program on tiled processors.

- **ILP.** The low-cost SON of tiled processors has enabled compilers to exploit ILP. ILP has rarely been converted into TLP in previous parallelizing compilers due to prohibitively expensive communication cost. Consequently, ILP has been mostly exploited in hardware, even in CMPs. Compared to such hardware-managed ILP, compiler-managed ILP has a clear advantage in scalability. While the amount of exploitable ILP in a superscalar processor is limited by the width of the processor, which is not likely to scale, the ILP of a tiled architecture scales as long as the available transistor count on a chip scales according to Moore's law.

- **Memory Parallelism.** In many applications, an intelligent memory instruction scheduling can significantly improve the performance by hiding cache miss latencies. Superscalars have to find such opportunities in their limited instruction windows with complex hardware support. With a tiled processor, however, a compiler can easily perform memory scheduling by carefully assigning memory operations across multiple cores and overlapping computation instructions. This advantage comes from the combination of distributed memory systems and powerful static compiler analyses.

- **Multiple Control-flow.** In tiled processors, each core has its own control unit. This can be exploited by concurrently executing multiple code regions. For example, two cores can simultaneously execute two independent loops. Branch prediction and speculative execution allow conventional monolithic processors to extract parallelism across code regions, but such parallelism is limited within only one particular control-flow. Indeed, if a branch misprediction is detected, all instructions after the branch are discarded regardless of its control dependences in such a processor. By carefully analyzing and scheduling instructions across larger scopes, the compiler in a tiled processor has potential to extract parallelism from multiple regions of a program, something which is not achievable in conventional processors.

5.1.2 Challenges

The goal of a parallelizing compiler is to reduce the execution time of a program by exposing thread-level parallelism. Many factors should be considered to perform high-quality parallelization, otherwise the benefit of parallelization could be limited or the parallelization could results in worse performance. Specifically, memory communication, locality, and load balancing must be considered.

- **Communication.** In superscalars, scalar operand communications are managed by hardware and the communication cost is free. However, tiled processors do not have the luxury of hardware managed communication. As a result, the compiler has to explicitly manage all the communications in a tiled processor. Communication management could have a substantial impact on the parallelized program execution performance not only because communication takes time but also because it creates synchronization points between threads, potentially lowering the utilization of cores. Even a few cycles of communication cost could negate the benefit of fine-grained parallelization. Therefore, communication cost must be taken into account in fine-grained parallelization.

- **Memory Locality.** Being one of the major factors in determining performance, memory locality has been extensively studied in parallel compilers. For example, [9] proposed memory locality optimizations for coarse-grained parallelization. In tiled processors, the memory locality problem is further complicated by fine-grained parallelization. For example, if a load instruction accesses data residing on a remote core, it incurs cache coherence traffic, increasing the instruction latency. Unless enough parallelism is exploitable, such an instruction should be placed on the core where the data is located to avoid cache coherence traffic.

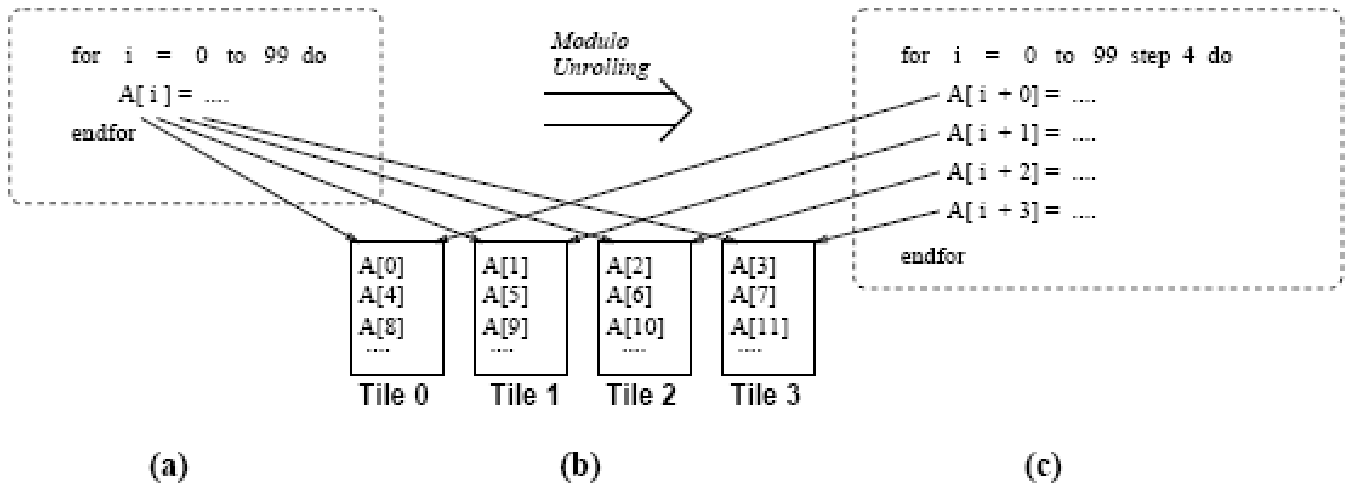


Figure 4: Example of modulo unrolling[2]

- **Load Balancing.** The execution time of a parallelized program is determined by the longest thread execution time in the system. Thus, it is critical to evenly distribute workload across available cores. Memory bandwidth load balancing as well as computation load balancing is important in tiled processors where memory systems are also distributed, especially in applications with high memory bandwidth requirements. Note that good load balancing can be achieved only when an accurate estimation of execution time is provided, so the estimation technique must consider various aspects of the thread execution that could affect the execution time. These factors include computation, memory accesses, and communication.

So far, we have discussed opportunities and challenges in the design of a parallelizing compiler for tiled multicore processors. In subsequent subsections, we will explore proposed parallelization techniques and examine how these opportunities and challenges are handled.

5.2 ILP and Memory Parallelism: RAWCC

5.2.1 Motivation

RAWCC was designed to exploit memory parallelism and instruction parallelism in the RAW processor. As the memory systems and functional units in the RAW processor are distributed across tiles, it is imperative to efficiently manage those distributed resources to expose parallelism. RAWCC consists of two steps. First it orchestrates memory accesses [2] then it exploits fine-grained ILP with its space-time scheduler [13].

5.2.2 Memory Parallelism

In tiled multicore processors, the management of memory systems is more complicated because of their distributed nature. Accessing data in a remote core typically takes more clock cycles compared to accessing local data due to the communication cost of memory networks. Doing every memory access in the same core avoids such a memory communication cost but doing so misses opportunities for memory parallelism. The compiler has to avoid communication cost while distributing memory accesses so that distributed memory systems are evenly loaded.

RAWCC performs equivalence class unification (ECU) in order to reduce inter-tile communication. ECU first builds equivalence classes, which unifies potentially conflicting memory accesses. The compiler gathers memory access conflict information by performing pointer analysis, and it assigns a core to each unified class so that all the operations in the same equivalence class are executed without paying communication cost. Although ECU efficiently avoids paying communication cost for scalar data, the compiler requires another technique to handle aggregate memory objects such as an array. If all memory accesses to an array had to be made on the same core as in ECU, both memory parallelism and other forms of parallelism would be severely restricted. This is because memory load instructions typically have multiple dependent instructions.

To extract memory parallelism from aggregate memory objects, RAWCC uses a technique called modulo unrolling. Modulo unrolling assumes a memory distribution pattern called low-order interleaving where consecutive elements of an array are interleaved in a round-robin manner across cores. The modulo unrolling exposes memory parallelism through an intelligent loop unrolling. The execution core for each memory operation is statically determined by a modulo operation to effectively utilize memory parallelism. Figure 4 shows an example of modulo unrolling. After the loop (a) is unrolled with the factor of four - this is decided by the analysis of modulo unroller - cores are assigned for all the four memory accesses in the unrolled loop. As a result, the loop execution is allowed to use four times the memory bandwidth compared to the single core execution. Modulo unrolling not only exposes memory parallelism but also increases ILP in loops. This exposed ILP is then exploited in the next phase of the compiler.

5.2.3 Space-time Scheduling

Once cores for memory access instructions are pre-assigned through ECU and modulo unrolling, the compiler schedules instructions across cores. The scheduler in RAWCC is called

a *space-time scheduler* since it determines both *when* and *where* each operation is executed. In contrast, conventional schedulers only care *when* an operation should be performed. To deal with the increased complexity, RAWCC divides the problem into two parts. It first decides where each operation should be executed (spacial scheduling) then fixes the schedule of instructions inside each processor core (temporal scheduling).

The spatial scheduler first detects ILP in each basic block. Although the basic block scope limits the amount of extractable parallelism, it also has a few advantages. First, it makes the management of control dependences easy since every core concurrently executes the same basic block of the program. To make every core follow the same control-flow, the control-flow of the single threaded program is mirrored in each thread. This is achieved by duplicating each branch instruction in every core and broadcasting branch decisions at run-time. Another advantage of basic block based scheduling comes in load balancing. Since all the instructions in the same basic block have the same weight in terms of execution frequency, the load for each thread can easily be estimated. This results in evenly loaded threads in the end.

After ILP is detected, the spatial scheduler partitions the program into multiple threads. This process is essentially to convert ILP into TLP. Thread partitioning consists of two steps. The compiler first makes clusters assuming an infinite number of cores using the DSC algorithm [22]. Multiple clusters are created only when the benefit of parallelization is large enough to offset the communication cost. Once clusters are built, they are merged so that the targeted number of well-balanced threads are created. Reasonably good load balancing can be achieved in the merging process by estimating the execution time of each cluster and the resulting partitions. After partitions are determined, communication instructions are appropriately inserted to respect data dependences across threads.

Once the spatial scheduler completes thread partitioning, a conventional greedy list scheduler is used for the temporal scheduling. The scheduler keeps track of a ready list for each core and fills the ready list with the highest priority instruction. The priority of an instruction is determined by its level and fertility. Level is defined to be its critical path length to an exit node and fertility is defined as the number of descendant nodes divided by the number of tiles. The list scheduler uses a weighted sum of both of these metrics.

5.2.4 Discussion

RAWCC successfully parallelizes most SpecFP benchmarks [21]. With the combined efforts to exploit both ILP and memory parallelism, RAWCC effectively converts distributed computation resources and memory resources into meaningful performance. However, RAWCC has a few limitations. First, it supports only low-order interleaving for memory parallelism, which is not necessarily the best data distribution for every application. Second, the multi-stage heuristic algorithm of RAWCC is sub-optimal. Although it is a reasonable strategy to break a hard problem into smaller ones, the resulting performance can suffer from the well-known phase-ordering problem. In fact, [14] improved the performance by using a method similar to simulated annealing.

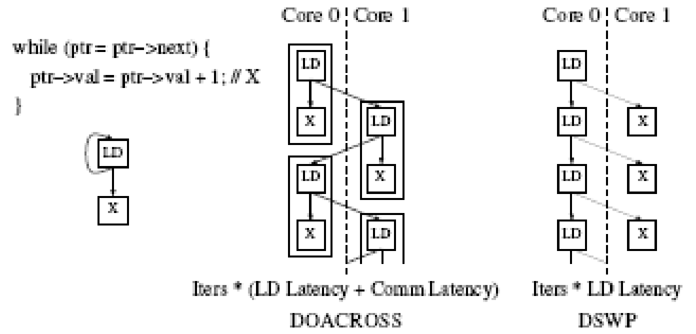


Figure 5: A motivating example of DSWP[17]

Finally, target applications that can benefit from RAWCC are limited. RAWCC performs well in DOACROSS loops of numerical applications exploiting memory parallelism and ILP through modulo unrolling, but not in general purpose applications where available ILP in a basic block is typically not high.

The Performance of RAWCC depends on the latency of the SON because threads are tightly coupled by various forms of dependences [20]. However, an increase in a queue size does not typically reduce the execution time in RAWCC. This is because resulting threads usually have cyclic dependences, creating many synchronization points. In fact, the restricted parallelism by cyclic dependences directly motivates the next parallelization technique, DSWP.

5.3 Pipeline Parallelism: DSWP

5.3.1 Motivation

Decoupled Software Pipelining (DSWP) [17] is another parallelization technique. Unlike RAWCC, which mainly targets ILP and memory parallelism, DSWP extracts fine-grained pipeline parallelism in loops, transforming a loop into multiple decoupled loops running on multiple cores. The motivation of DSWP is shown in Figure 5. In the typical DOACROSS parallelization, each loop iteration is assigned to available cores in a round-robin fashion. However, it causes tight couplings between loop iterations, creating a synchronization point for each loop-carried dependence pair. In Figure 5 the critical path of this loop is the chain of LD instructions. When a synchronization point is created in the critical path as in Figure 5 (b), it increases the total execution time.

The key benefit of DSWP is to avoid communication in the critical path, which reduces the total execution time. In most loops, the critical path of the loop involves loop-carried dependences. To avoid worsening the critical path, DSWP simply puts all instructions forming a loop-carried dependence in the same core. For example, all the load instructions in Figure 5 are scheduled on Core 0.

5.3.2 DSWP Thread Partitioning Algorithm

DSWP uses the PDG discussed in Section 2.1 in the management of parallelism. Figure 6 (b) shows the PDG built from the program in Figure 6 (a). In a PDG, a strongly connected component (SCC) represents a code region with loop-carried dependences. DSWP merges such an SCC into

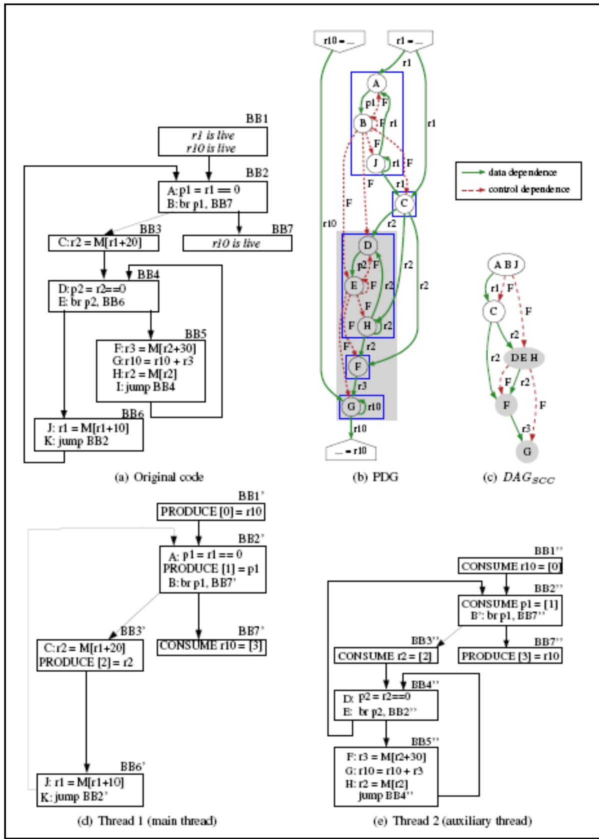


Figure 6: A DSWP example[17]

a single node, building a directed acyclic graph (DAG) as shown in Figure 6(c). Finally, SCCs are grouped to form the targeted number of threads in such a way that threads have only acyclic dependences. The fact that threads are free from any acyclic dependence guarantees the decoupled execution in the parallelized threads. The thread partitioning process of DSWP resembles that of RAWCC. First it makes many parallelization units (clusters or SCCs) then it merges them based on their loads. The quality of DSWP parallelization is almost as good as manual parallelization in SPEC benchmarks, which are difficult to parallelize [17].

5.3.3 *Discussion*

DSWP offers a few advantages over RAWCC. First, it can extract parallelism across basic blocks. Although the scope is still restricted within a loop, it can find and exploit parallelism that RAWCC cannot. This is why DSWP could perform well on general-purpose programs where ILP inside a basic block is limited. Second, DSWP is likely to outperform RAWCC in many applications by minimizing the execution time of the critical path. Finally, by decoupling thread executions, DSWP shows superior inter-thread communication latency tolerance. Instead, its performance is affected by the queue size because a longer buffer queue provides more tolerance at the execution time variance of each pipeline stage in DSWP.

DSWP has several problems, specifically in applicability, load balancing, and scalability. Most importantly, DSWP is not always applicable. The number of SCCs in a loop could be too small or the resulting threads might be too unbalanced to offset parallelization overhead. Another challenge of DSWP is load balancing. Since DSWP extracts parallelism across the basic block boundary, each SCC is likely to have a different execution weight at run-time depending on its control dependences. Therefore, DSWP has to calculate expected workload for each SCC by considering instruction latencies as well as the execution profile weight of the SCC. The final problem of DSWP is its scalability. In DSWP, the execution time of a loop is determined by the longest execution time of all the pipeline stages. Consequently, once thread partitioning is performed as such that the critical thread consists of only the critical SCC, the use of an additional core cannot bring any performance improvement in DSWP.

5.4 Harness Multiple Control-flow: GREMIO

5.4.1 *Motivation*

GREMIO (Global REgion Multi-threaded Instruction Orchestrator) [16] is another compiler parallelization technique. In overall, GREMIO is similar to RAWCC in many aspects, except the scope of parallelism extraction. Unlike RAWCC which enforces the same control-flow in all the threads, GREMIO allows each core has its own control-flow with its own control unit. It means different regions of a program can be concurrently executed, not restricted to the basic block or the loop boundary. GREMIO uses a variant of hierarchical PDG in the detection and management of parallelism across different regions of a code; its control-flow aware list scheduler performs thread partitioning and scheduling by traversing the PDG.

5.4.2 *GREMIO Algorithm*

GREMIO first builds the PDG of an input program as discussed in Section 2, then it converts the created PDG into HPDG (hierarchical program dependence graph). HPDG is a specially designed PDG for clustering purposes in GREMIO, and it is different from the conventional PDG in two aspects: 1) each loop nest is coalesced to a single node, making the graph hierarchical, 2) all loop-carried dependences are discarded inside a loop. The resulting HPDG is acyclic, so GREMIO can use a conventional clustering algorithm applicable to acyclic graphs. In fact, GREMIO uses the DSC [22] algorithm, which is also used in RAWCC. Although GREMIO disregards loop-carried dependences in the clustering step, all discarded dependences are satisfied in its scheduling step to guarantee the correct execution of the input program. Figure 7 (a) shows the HPDG resulting from Figure 1, and Figure 7 (b) shows HPDG after clustering. The clustering decisions are all respected in the scheduling by placing all the nodes in a cluster in the same thread.

Compared to local list schedulers which extract parallelism only within a single basic block, GREMIO list scheduler has an additional complication that results from handling control-flows. While the distributed control units in a tiled processor enable the concurrent execution of multiple code regions, it requires control-flow aware scheduler.

To deal with the control-flow management, GREMIO defines control relations between two HPDG nodes X and Y as follows:

- **Control Equivalent**, if both X and Y are simple nodes with the same input control dependences.
- **Mutually Control Exclusive**, if the execution of X implies that Y does not execute, or vice-versa.
- **Control Conflicting**, neither of the above hold true.

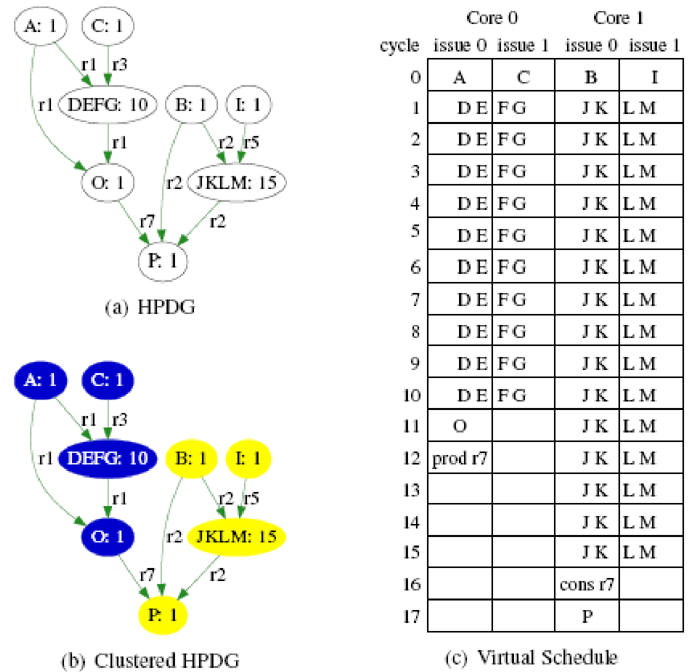
In Figure 7 (a), for instance, nodes A, B, C, I, O, and P are control equivalent; nodes DEFG and JKLM are control conflicting with every other node; no mutually control exclusive node pair exists in the given example.

At a specific cycle time, a node in the HPDG becomes eligible for scheduling when it has the highest priority among nodes whose dependences are all satisfied. Of course, communication cost has to be considered for inter-core data dependences. Similar to other conventional list schedulers, the priority of a node is determined by its distance to a leaf node. Suppose the scheduler tries to assign a node X to a functional unit Y in core Z at time slot T. The control-flow information is used in the following manner: 1) if a control equivalent node is already assigned to Y, X cannot be scheduled in the function unit Y at time T, 2) if a mutually control exclusive node is assigned to Y, X can be scheduled to the same functional unit Y at T, and 3) if any control conflicting node is already scheduled in core Z at time slot T, X cannot be scheduled in core Z at time slot T. The restriction for the third case originates from the limitation that a processor core can execute only one control-flow at a time, while two control conflicting nodes might have different control dependences. Figure 7 (c) shows the schedule of our running example in a tiled processor with two-cores where each core has two functional units. Two independent loops - DEFG and loop JKLM - will be concurrently executed in the example.

The described scheduling algorithm raises a problem in handling loop nodes. If the scheduler naively assigns all the instructions in a loop node to the same core, potential parallelization opportunities lurking in a loop will not be exploited. Since loops are important targets in any parallelizing techniques, it is highly desirable to execute a loop node using multiple cores. The problem becomes more complicated by the existence of nested loops, which makes it a recursive problem. To solve this problem, GREMIO uses dynamic programming. From leaf loop nodes in a loop nest, GREMIO estimates the execution time of each iteration with various number of threads in a bottom-up manner. When the algorithm completes processing the root node, the algorithm produces the estimated number of threads that is likely to minimize the execution time of the program. This dynamic programming does not necessarily result in the optimal solution since a scheduling for a leaf loop is already NP-hard problem but the authors claim that it achieves significant speedup in various benchmarks [16].

5.4.3 *Discussion*

Control-flow substantially limits the amount of exploitable parallelism in general purpose programs [12], but it has been hard to exploit parallelism across different code regions. Single processor systems have only one control unit so they



cycle	Core 0 issue 0	Core 0 issue 1	Core 1 issue 0	Core 1 issue 1
0	A	C	B	I
1	D E	F G	J K	L M
2	D E	F G	J K	L M
3	D E	F G	J K	L M
4	D E	F G	J K	L M
5	D E	F G	J K	L M
6	D E	F G	J K	L M
7	D E	F G	J K	L M
8	D E	F G	J K	L M
9	D E	F G	J K	L M
10	D E	F G	J K	L M
11	O		J K	L M
12	prod r7		J K	L M
13			J K	L M
14			J K	L M
15			J K	L M
16			cons r7	
17			P	

Figure 7: Operation of GREMIO on the example from Figure 1[16]

can track only a single control-flow at a time. Conventional parallel systems can run multiple code regions at a time but their expensive communication cost severely restricts the applicability. With tiled multiprocessors, which are equipped with both multiple control units and low-cost on-chip networks, it is finally feasible to benefit from the parallelism across code regions. DSWP also overcomes the basic block boundary but its parallelism extraction is still restriced within a loop boundary. For instance, DSWP cannot simultaneously execute two independent loops in the code. In contrast, GREMIO does not have any restrictions on parallelism extraction.

The interesting part of GREMIO is load balancing. Suppose GREMIO creates two threads executing two independent loops. If one loop terminates much earlier than the other, it could be better to execute one loop at a time using two cores rather than globally extracting parallelism. In other words, global scheduling does not necessarily outperform local scheduling if good load balancing is not provided. GREMIO solves this problem by using profiling but doing so makes the efficiency of a program dependent on its input. The use of run-time information with dynamic thread spawning can be an alternative solution for this problem. We will discuss this issue in more detail in Section 6.1.

5.5 Summary

So far, we have explored three parallelization techniques in tiled processors. Table 1 summarizes these techniques. All three techniques can exploit the fine-grained parallelism enabled by the low-cost SON in tiled processors. Unlike conventional parallelizing compilers, which can extract only coarse-grained parallelism, RAWCC focuses on ILP and memory parallelism of loops. This allows for significant speedup

	RAWCC	DSWP	GREMIO
Parallelism Type	ILP, Memory	Pipeline	ILP
Parallelism Scope	Basic Block	Loop	Procedure
Intermediate Representation	CFG-based	PDG-based	PDG-based
Load Balancing	No Profiling Needed	Profiling-based	Profiling-based
Applicability	General	Limited	Limited
Scalability	Good	Limited	Good
Sensitive to Queue Latency	Yes	No	Yes
Sensitive to Queue Length	No	Yes	No

Table 1: Comparison of RAWCC, DSWP, and GREMIO

in numerical applications without relying on complicated loop transformations. GREMIO is similar to RAWCC in many ways but it can extract parallelism across basic blocks, exploiting distributed control units in tiled processors. Finally, DSWP creates multiple pipeline stages (threads) from a loop to exploit pipeline parallelism.

Since parallelization is a challenging task, all three techniques divide the parallelization task into multiple sub-tasks. For example, both RAWCC and GREMIO use the DSC algorithm [22] to create parallel code regions then apply a greedy list scheduler for load balancing. Unfortunately, this multi-phase solution can suffer from the phase ordering problem, leaving room for improved performance.

From the three fine-grained parallelization techniques we have examined, we can see the tradeoff between the scope of parallelism extraction and load balancing. In RAWCC where parallelism is extracted only in the same basic block, load balancing is easy because each instruction has the same weight. However, as the scope of parallelism extraction is extended, load balancing gets more complicated, especially when code regions with variable execution time are involved. DSWP and GREMIO solved the load balancing problem by using profiling data.

Another notable point is the impact of the SON performance on parallelization. RAWCC and GREMIO are sensitive to the latency of the SON in most cases. This is expected because the execution time is directly affected by the SON performance as long as the critical path involves any communication. Unlike RAWCC and GREMIO, each thread of DSWP is a pipeline stage and the execution time of a program is determined by the longest pipeline stage. Therefore, the critical thread execution time will be affected only when the increased latency of the SON delays the availability of data needed in the critical thread. This scenario is not likely, considering that non-critical threads typically have smaller execution times. In contrast, the size of the communication queue can have a significant impact in DSWP since a longer queue can tolerate more variability in the execution time caused by variable latency code regions and unexpected run-time events like cache misses. In RAWCC and GREMIO, however, the tightly coupled nature of threads make them very insensitive to the communication queue length. Note that RAWCC and GREMIO can accidentally produce DSWP-style threads without any cyclic dependence. In such a case, the performance will be sensitive to the length of a queue rather than the latency of the communication [16].

According to performance comparisons performed by researchers [16, 23], no single technique is superior to the others. The best way to parallelize a code region depends on the characteristics of the code region and the type of the application. Therefore, it is important to understand the advantages and disadvantages of each technique and apply the most suitable one.

6. FUTURE DIRECTIONS

We have examined three compiler parallelization techniques in the previous section. While providing good performance gains, they are by no means optimal in exploiting parallelism in a program. Since tiled microprocessors are relatively new, we expect more innovative schemes will be proposed in the near future. In this section, we propose a few possible future research topics in this area.

6.1 Use of a Runtime Thread Scheduler

All the techniques we have examined use static load balancing. However, static load balancing raises a problem for code regions with variable execution times such as unbounded loops. Since this kind of uncertainty is dependent on program inputs, the use of run-time information can alleviate the problem. Specifically, the use of dynamic thread spanwing in addition to a run-time scheduler can improve the core utilization and reduce the program execution time. The Cilk [3, 8] project is a good example. Cilk adds a few keywords to the C programming language, allowing programmers to describe task-level parallelism in the program. At run-time, Cilk dynamically spawns threads and uses a work-stealing scheduler to achieve high processor utilization. Although Cilk was developed for conventional multiprocessor systems where only coarse-grained parallelization is feasible, the technique can also be applied to tiled multicore processors.

6.2 Locality-aware DLP Exploitation

Among compiler techniques we have examined, only RAWCC provides a DLP exploitation framework with modulo unrolling. Although modulo unrolling can be generally applicable, it does not necessarily bring the best results. Modulo unrolling assumes a specific memory distribution called low-order interleaving for every application. However, low-order interleaving might not be suitable for some applications. It is possible that low-order interleaving can destroy well-managed parallelism in the single threaded program. In other words, modulo unrolling enables the exploitation of memory parallelism at the possible expense of losing memory locality. Since the destruction of memory locality could negate the benefit from parallelization, it introduces the need of a locality-aware DLP exploitation framework.

One possible alternative to the modulo unrolling is having pre-designed templates for frequently occurring data patterns. In fact, High Performance Fortran [11] (HPF) has a few keywords specifying the data distribution patterns. In HPF, three data distribution types are supported for each array dimension: * for no distribution, BLOCK for block distribution, and CYCLE for cyclic distribution. With memory distribution information, a compiler is likely to achieve better parallelization.

6.3 Instruction Privatization

All three compiler techniques we have examined assume only one core is used to execute a specific instruction. However, the duplication of computation could break dependences and save communication cost. If the cost of duplicated computation is less than the communication cost, it is better to avoid communication by executing the same instructions on multiple cores.

We call this technique *instruction privatization* since it is similar to variable privatization technique [11] in the sense that both techniques duplicate a resource (functional unit or storage) to break dependences. In addition, it also resembles the idea of rematerialization in register allocators. In a register allocator, a register spilling can be avoided if the generation of a register value does not cost as much.

7. CONCLUSION

Tiled multicore processors were developed to provide scalable performance from ever increasing on-chip transistor counts. To maximize the benefit, however, it requires extensive supports from a compiler for tasks such as exploiting parallelism, communication management, memory locality management, and control-flow management.

In this work we have investigated various compiler parallelization techniques developed for tiled multicore processors. These parallelization techniques are distinguished from conventional parallelizing techniques in that they exploit fine-grained parallelism with the support of low-cost SONs in tiled multicore processors. Although the investigated techniques showed their effectiveness in some benchmarks, there are still many future research topics in this area. With the upcoming era of multicore processors, the applicability and effectiveness of these fine-grained parallelization techniques are likely to have a significant impact on the design of future microprocessors.

Acknowledgements

The author would like to thank Saturnino Garcia and Anshuman Gupta for comments on drafts of the paper.

8. REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: a compiler-managed memory system for raw machines. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 4–15, Washington, DC, USA, 1999. IEEE Computer Society.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[4] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP*, pages 836–844, 1986.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[10] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 146–157, New York, NY, USA, 1988. ACM.

[11] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[12] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 46–57, New York, NY, USA, 1992. ACM.

[13] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 46–57, New York, NY, USA, 1998. ACM.

[14] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 111–122, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural*

support for programming languages and operating systems, pages 2–11, New York, NY, USA, 1996. ACM.

[16] G. Ottoni and D. August. Global multi-threaded instruction scheduling. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–68, Washington, DC, USA, 2007. IEEE Computer Society.

[17] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[18] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers. The wavescalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4, 2007.

[19] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[20] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):145–162, 2005.

[21] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[22] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.*, 5(9):951–967, 1994.

[23] H. Zhong, S. A. Lieberman, and S. A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, DC, USA, 2007. IEEE Computer Society.