Automatic Partitioning of Signal Processing Programs for Symmetric Multiprocessors

Chris J. Newburn and John Paul Shen
Department of Electrical and Computer Engineering
Carnegie Mellon University
{newburn, shen}@ece.cmu.edu

Published in the Proceedings of Parallel Architectures and Compilation Techniques, October 20-23, 1996

Automatic Partitioning of Signal Processing Programs for Symmetric Multiprocessors

Chris J. Newburn* and John Paul Shen*
Department of Electrical and Computer Engineering
Carnegie Mellon University
{newburn, shen}@ece.cmu.edu

Abstract

*Symmetric multiprocessor systems are increasingly common, not only as servers, but as a vehicle for executing a single application in parallel in order to reduce its execution latency. This paper presents PEDIGREE, a compilation tool that employs a new partitioning heuristic based on the program dependence graph (PDG). PEDIGREE creates overlapping inter-dependent threads, each executing on a subset of the SMP's processors that matches the thread's available parallelism. A unified framework is used to build threads from procedures, loop nests, loop iterations, and smaller constructs. PEDIGREE does not require any parallel language support; it is a post-compilation tool that reads in object code. The SDIO Signal and Data Processing Benchmark Suite has been selected as an example of real-time, latency-sensitive code. Its coarse-grained data flow parallelism is exploited by PEDIGREE to achieve speedups of 1.56*x*/2.11*x *(mean/max) and 1.61*x*/2.60*x *on two and four processors, respectively. There is roughly a 15% improvement over existing techniques that exploit only data parallelism. By exploiting the unidirectional flow of data for coarse-grained pipelining, the synchronization overhead is typically limited to less than 6% for synchronization latency of 100 cycles, and less than 2% for 10 cycles.*

1. Introduction

Symmetric multiprocessor (SMP) systems constructed from high-end superscalar processors are increasing common. Most high-end server systems employ small-scale SMP, and this is anticipated to spread even to desktop systems. This trend is driven by several developments in commercial microprocessors: higher performance, falling costs, availability of built-in multiprocessor support, and a higher level of integration in chip packaging.

Parallelism is typically exploited by SMPs in three ways: by running separate programs in parallel (multiprogramming), by requiring the programmer to specify how a single program's code and data should be partitioned, or by exploiting parallelism only among loop iterations. For some applications, the only way to reduce execution latency enough to meet critical timing constraints is to exploit parallelism in a more general way than provided by current compiler technology. As a result, the programmer must partition and schedule code by hand.

*This research was supported by ONR contract numbers N00014-91-J-1518 and N00014-96-1-0347. We would like to thank the Pittsburgh Supercomputing Center for use of their Alpha systems.

One such application domain is on-board signal processing. The Strategic Defense Initiative Organization (SDIO) Signal and Data Processing Benchmark Suite [Nic91] is a set of kernels drawn from aerospace and military systems that perform functions such as radar and sonar signal processing. These kernels are CPU intensive. Because they have real-time scheduling demands, minimizing latency is critical. Traditionally, these applications run on custom-designed distributed-memory systems that have up to hundreds of processors operating in a coarse-grained data flow fashion, as illustrated in Figure 1. The application is usually partitioned by hand into stages that are spread across the processors. Code is generated separately for each processor and communication and synchronization among these processes is explicitly specified by the programmer. Frequently, the poor quality or unavailability of compilers for the specialized processors used requires hand-coding in assembly language.

An alternative to this approach is to replace a large number of potentially specialized processors in these systems with a much smaller number of SMP systems. Projected trends in advanced CMOS technology indicate the availability of small-scale SMPs on a single chip or MCM at relatively low prices in the near future. It is inevitable that aerospace and military on-board signal processing systems will need to leverage the price/performance benefits of such commercial SMP systems. There is a crucial need for compilation tools that can take existing application software, possibly in the form of compiled object code, and partition and schedule it for efficient execution on such systems. Such tools will facilitate the porting of on-board signal processing sequential programs for parallel execution on these modern high-performance SMP systems.

The specific focus of this paper is to investigate the feasibility of automatically parallelizing on-board signal and data processing applications for parallel execution on commercial SMPs. We present the PEDIGREE compiler,

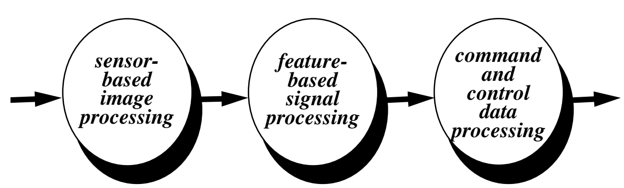

Figure 1. High-level view of typical on-board signal processing

which reduces the latency of a single program on an SMP without language support or programmer intervention, and exploits parallelism among more than just loop iterations and explicitly-parallel constructs. Section 2 shows how our approach relates to previous work. Section 3 presents PEDIGREE, a post-pass compiler which performs scheduling and partitioning for a multiprocessor. Section 4 introduces a novel partitioning heuristic employed by PEDIGREE. Section 5 presents results which demonstrate PEDIGREE's effectiveness in partitioning and scheduling the SDIO benchmarks on an SMP, in a way that is tolerant of significant synchronization latency. Conclusions and further work are discussed in Sections 6 and 7.

2. Related Work

There are three traditional approaches to running programs in parallel on SMPs [Ost87,Thi92]: multiprogramming, exploiting control parallelism, and exploiting data parallelism. In multiprogramming, independent programs run on different processors to increase a system's overall throughput. The second approach exploits control parallelism by executing code with different functionality in parallel on different processors. Pipelining [Thi92] and procedure calls within cobegin/end [Ari82] are examples. In this context, tasks are typically executed as processes on different processors, and scheduled by the operating system. Scheduling overhead often requires that the granularity of these processes be at the procedure level. Usually, the programmer must specify this kind of parallelism with special language features [Ari82, SH86] or library calls [SFG91, TRG+87, GBD+91]. There has been some related work in scheduling pre-partitioned task graphs [ERA91, GY92, GP88, YFGS95] and in partitioning graphs for parallel execution [GP94, RNSB94].

The third approach exploits data parallelism by performing the same functionality in parallel on different data sets. This type of parallelism is exploited almost exclusively among loop iterations. The amount of synchronization overhead determines the granularity. The programmer can specify this type of parallelism with language support. High Performance Fortran (HPF) [Hig93] supports data parallelism with the *FORALL* statement, which expresses assignments to sections of arrays, and the *INDEPENDENT* directive, which asserts the absence of sequentializing dependences. Some other examples of language support include parallel loop constructs such as doall and doacross [Cyt87], intrinsic array primitives, and complex communication primitives such as parallel prefix [Thi92]. Some compilers, such as [AALL93, Arn82, THK93], find data parallelism in loops automatically. Data is often mapped across processors by the programmer, perhaps with some compiler assistance [THK93, CMZ92, Hig93, GOS94, BCG+94, KP96].

Our compiler, PEDIGREE, automatically parallelizes a single program across multiple processors. The key differences from previous work pertain to the degree of automation, generality, granularity, the overlap of inter-dependent code, and flexibility in mapping available parallelism to subsets of processors. PEDIGREE is able to parallelize code automatically, and does not depend upon explicit parallel language features or upon direction from the programmer. It is a post-pass optimizer that reads compiled object code as its source language. PEDIGREE is general in that it features a unified framework based on the program dependence graph (PDG) for automatically exploiting control and data parallelism at different granularities. The PDG is used to identify regions of the program that can be executed in parallel: procedure calls, loop nests, loop iterations, conditionals, and smaller constructs. A partitioning heuristic is introduced in this paper that selects regions to overlap (execute in parallel) so as to optimize the overall execution latency. Overlapped regions may be data-interdependent, unlike [ABC+88, GP88]. Resource usage is tailored to available parallelism in that overlapped code may be assigned to any subset of processors. This differs from PTRAN [Sar91], which either executes a loop in parallel across all processors, or executes it serially. These features make PEDIGREE unique in its ability to exploit parallelism across a range of granularities in a unified way, without requiring special language features or even the availability of high-level language source code. Our previous papers introduced our version of the PDG and explored the PDG's potential to expose parallelism [NNS94], and developed techniques for balancing the exploitation of different kinds of parallelism within loops [NHS93]. This work focuses on partitioning of code for execution on SMPs.

3. PEDIGREE Compiler

There are four distinctive features of PEDIGREE: it is post-pass, retargetable, based on the program dependence graph (PDG), and uses a novel approach to partitioning and scheduling for a multiprocessor. A discussion of PEDIGREE's partitioning and scheduling is deferred until the next section. Figure 2 provides an overview of the compilation flow. The compiler's input is an assembly language program, from which it generates several program representations for analysis and parallelization. The PDG is the most important of these representations, since it is used as the basis for program partitioning and scheduling. PEDIGREE then generates code for each processor as specified by the partition, and inserts branches and synchronization as necessary. The resulting parallel program is simulated on a timing and functional simulator to analyze perfor-

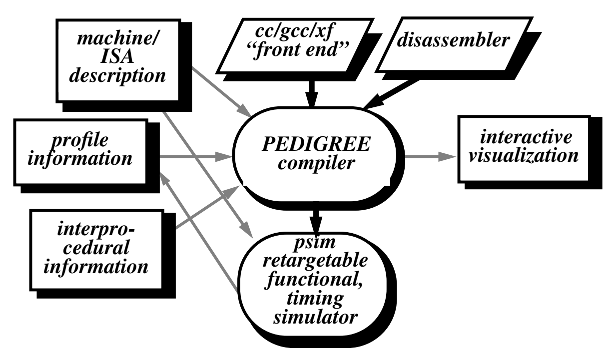

Figure 2. PEDIGREE (PDG-based Retargetable Evaluation Environment) tools suite.

mance and verify correctness.

There are several motivations for the post-pass approach adopted by PEDIGREE. The first is that it can leverage existing front-end compilers to generate assembly code from high-level languages. Thus, PEDIGREE need not implement many of the standard optimizations, and it can support compilation from several different high-level languages. Second, PEDIGREE can be used to optimize legacy assembly code for which the high-level source code no longer exists. Third, this tool can be leveraged as a foundation for object-to-object code transformation and potentially even binary translation of legacy code. Granted, some of the information provided by high-level language source code is lost at the assembly language level. However, with the use of a symbol table and possibly interprocedural information, the impact of this loss of information has been found to be minimal in most cases.

Retargetability is a key feature of PEDIGREE. Only the program's structure and semantics are directly manipulated by the PEDIGREE compiler and simulator. Specifics of the ISA are separately encapsulated. A set of files is used to describe ISA syntax and semantics, and machine characteristics such as latencies, hazards, issue policies, communication latencies, memory configuration, and synchronization mechanisms. PEDIGREE's retargetable post-pass approach can be leveraged to compile code for different architectures and implementations.

The PDG [NNS94] provides the foundation for partitioning and scheduling code in PEDIGREE because it makes finding and representing code to execute in parallel easier. The complexity of constructing a PDG from the procedure's control flow graph (CFG) is dominated by the construction of the control dependence graph, which is linear using the techniques of [JPP94]. The PDG represents the hierarchical control dependence structure of a procedure. It is hierarchical in the sense that each construct (such as a loop or conditional) is represented by a single node **R**, and all of its constituent smaller constructs (such as nested loops and conditionals) are descendants of **R**. For example, in Figure 3, the node P3 in the PDG represents the conditional B3, B4, B5. Nodes that correspond to the basic blocks nested within the conditional, C4 and C5, are children of P3. The basis for these hierarchical relationships is control dependence. All nodes that are immediately control dependent on node **X** are children of **X**. If the execution of node **R** depends on the outcome of a branch in **X**, then there is a directed arc that leads from **X** to **R**. This arc is labeled with the branch outcome that leads to **R**, such as *T* or *F.*

Each PDG node, along with the subgraph rooted at that node, represents a code construct or code fragment that is called a *region.* There are four principal node types in the PDG: *Code* nodes, *Call* nodes*, Predicate* nodes, and *Multi-Pred* nodes [NNS94]. Each one represents a different kind of region, as differentiated by its control dependence characteristics. Code nodes are leaf nodes that represent operations in a basic block. Nothing can be control-dependent on a Code node. Call nodes are leaf nodes that represent procedure calls. Predicate nodes represent conditional branches. Nodes that are control dependent on the conditional branch associated with a Predicate node are children of that node. Finally, a MultiPred node groups together nodes with a common set of control dependence ancestors. The most common instance of such a node is a *Loop* node, which has control dependences on the conditions that led to the loop's execution, and another control dependence on the Predicate node that represents the conditional branch that selects whether the loop is exited or repeated. The PDG has a root node, called the *Proc* node, that represents the whole procedure. It is a special kind of Predicate node, because the execution of the whole procedure is dependent on whether the procedure is entered. The differentiation among node types allows each type of region to be treated according to its particular characteristics.

Figure 3 illustrates a code fragment from SPICE and its CFG and PDG representations. The procedure has two conditionals, one loop, and some other basic blocks. In the PDG, the loop is represented by Loop node L2, and conditionals associated with basic blocks B3 and B6 are represented by Predicate nodes P3 and P6. An important feature of the PDG, illustrated in Figure 3c, is that available control parallelisms are made obvious. Nodes that have exactly the same set of control dependences are control equivalent. For example, nodes {C1, C3, P3, P6, C8} are control equivalent, but {C3, C4} are not. L2 is always executed when {C1, C3, P3, P6, C8} are, because it shares the same immediate ancestor via an arc with the same branch condition. For brevity, the set of nodes that are reached via an arc from a common parent that is labeled with the same branch condition shall be called control-equivalent for the rest of the paper, even though in instances like L2, this is not strictly the case. Control-equivalent nodes are good candidates for parallel execution, since if one is executed, all the others are as well. The loop and conditionals in this example are completely data independent, so they can all be executed in parallel. The PDG is useful for exposing control parallelism because it explicitly represents control dependence.

The hierarchical abstraction of a program's code constructs forms the basis for overlapping code fragments that range in granularity from a few instructions in a basic block to collections of loops, conditionals, procedure calls, and basic blocks. The PDG's explicit representation of control dependences makes finding candidates for parallel execution easier.

4. Program Partitioning and Scheduling

The target machine for this work is a symmetric multiprocessor (SMP), in which each processor can be a superscalar processor. Processors communicate via a shared memory. Synchronization primitives are used to ensure that inter-processor data dependences are maintained as data is passed from one processor to another. The number of processors (*N*) and the number of issue slots (*I*) in each superscalar processor may vary for different machine configurations. This type of machine configuration allows the exploitation of two types of parallelism: fine-grained parallelism within each superscalar processor, and coarse-grained parallelism across different processors.

```
    NUMEL=0
    DO 360 I=1,18
360 NUMEL=NUMEL+JELCNT(I)
    NUMTEM=MAX0(NUMTEM-1,1)
    IDIST=MIN0(IDIST,1)

B1: numel = 0
    i = 1
B2a:jelcnt_i = load jelcnt,i
    numel = numel + jelcnt_i
B2b:i = i + 1
    b (i <= 18) B2
B3: numtem’ = numtem - 1
    b (numtem’ ≤ 1)B5
B4: numtem = numtem’
    br B6
B5: numtem = 1
B6: b (idist ≤ 1) B8
B7: idist = 1
B8: ...
```

Figure (a) FORTRAN and pseudo code for a fragment of SPICE

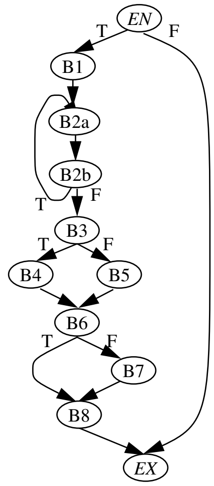

Figure (b) The CFG for the fragment

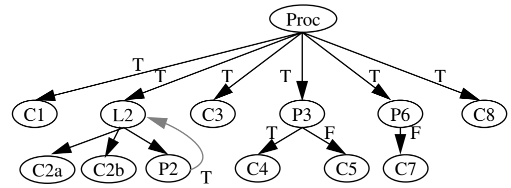

Figure (c) PDG for the fragment

Bi	Basic block i in CFG
Ci	*Code* node for CFG node Bi
Pi	*Predicate* node for CFG node Bi
Li	*Loop* node
Proc	*Proc,* the procedure node
EN	Procedure entry
EX	Procedure exit
	Control dependence induced by loop back arc

Figure 3. (a) FORTRAN example from SPICE, and its (b) CFG and (c) PDG.

In order to exploit coarse-grained parallelism across the different processors, each procedure of the program must be partitioned. The term partition is usually used to denote the division of a program into tasks that can be scheduled for concurrent execution. For this paper, we incorporate elements of the schedule into the definition of a partition. A *partition* specifies what regions are to be executed in parallel, which processors they are assigned to, and how those regions are ordered within each processor. The goal of the partitioning and scheduling process is to overlap regions on different processors so as to minimize the overall program latency, while taking synchronization and communication overhead into account.

Figure 4 shows an execution record for two possible partitions of the code in Figure 3. The program is spread across two processors, and the figure shows how regions have been assigned to processors and the relative ordering of the regions on each processor. In Figure 4a, the loop body is spread across two processors, and conditionals P3 and P6 are executed in parallel on processors one and two. Figure 4b shows an alternate partition, in which the loop is executed on processor one, and the two conditionals are executed serially on processor two.

The number of processors that each region can be executed on ranges from one to the total number of processors in the SMP. The selected number of processors is tailored to suit the balance of available parallelism in that region and in the rest of the program. Each Code node is executed on a single processor. But nodes representing larger constructs like loops and conditionals may have the regions represented by their children executed in parallel on multiple processors. The term *width* is used to denote the number of processors that a region is executed on. The possible widths are limited to powers of two. Selection of this width depends on the latency of each child region for each possible width, on dependence latencies among child regions, and on the synchronization overhead. The *dependence latency* from region *A* to region *B* is defined as the number of cycles that *B* must be delayed with respect to the start of *A* so that *B* does not have to wait for any data from *A*. There is no coarse-grained parallelism available within P6 in Figure 3. Hence the latency of executing P6 on two processors is no better than on one processor. The loop, however, can be spread across multiple processors. The partition that yields the execution record in Figure 4b may be a good choice if the loop and conditionals P3 and P6 are data independent, and if the serial latency of the loop is roughly equal to that of the two conditionals. If the loop’s execution latency is much longer than that of the conditionals, and the loop can be spread across multiple processors without incurring too much synchronization overhead, then the partition that leads to the execution record in Figure 4a may be a better choice.

4.1. Using the PDG for Partitioning and Scheduling

The PDG’s structure forms an excellent basis for finding what can be executed in parallel and for representing the resultant partition. The obvious candidates for non-speculative parallel execution are control-equivalent regions. Conveniently, these are represented by siblings of a common parent in the PDG that are reached via the same branch conditions. Thus specifying a partition for each set of control-equivalent children specifies the entire partition.

The PDG is modified to maintain partition information in three ways. First, a new type of node, a *Par* node, is inserted to group together control-equivalent children that are spread across multiple processors. For each region that is executed on multiple processors, represented by node **N**, and for each set of control-equivalent children of **N** (e.g. one for T and one for F for a Predicate), a Par node is inserted so as to be a child of **N** and a parent of that set of control-equivalent children. Additional Par nodes may be inserted to form a tree of subpartitions, with lower levels in the tree grouping together children that are executed on a progressively smaller subset of processors. Each Par

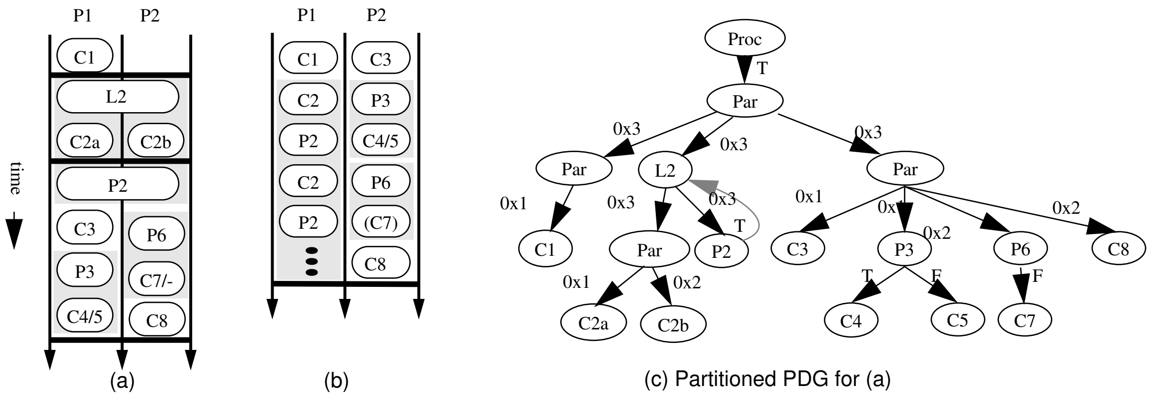

Figure 4. Partitioning example, showing the execution record for two possible partitions of the PDG for processors *P1* and *P2*. The regions of the PDG are outlined in dashed boxes. Control flow barriers are shown with dark horizontal lines. (c) shows the partitioned PDG. The processors are specified with a mask: 0x1, 0x2 and 0x3 mean processor one, processor two, and processors one and two, respectively.

node is marked with the number of processors that its children are assigned to, i.e. its width. Second, arcs to those children are annotated with a mask which specifies processor(s) they execute on. Third, those arcs are ordered to indicate the order in which the children execute.

Figure 4c shows the partitioned PDG that corresponds to Figure 4a. The regions represented by the Proc node and the Loop node are executed on multiple processors, so these nodes each have a Par node child. Regions represented by nodes C1 and {C3, P3, P6, C8} are each executed on a smaller number of processors than the loop is, so additional Par nodes are added to represent those subpartitions as well. This tree of Par nodes facilitates the synchronization of processors in the transition from one subpartition to the next. In Figure 4a, a barrier must be inserted at the end of the subpartition that includes C1 so that the loop starts executing at the same time on processors one and two. Likewise, processors must be synchronized before the return at the end of the procedure and at the end of each loop body, before the exit conditional. Each of these points is at the end of a subpartition, where barriers must be inserted as shown in Figure 4a. So, Par nodes are used to indicate where a synchronization barrier should be inserted.

4.2. Partitioning and Scheduling Heuristic Design

The primary goal of the partitioning and scheduling heuristic is to minimize the overall execution latency. There are four specific objectives for the heuristic: (1) identify regions that can be executed in parallel without speculation, (2) restrict the maximum scope of what can be parallelized to limit computational complexity, (3) localize trade-off decisions, and (4) tailor utilization of machine resources to available parallelism in each region

The previous subsection notes that control-equivalent siblings make good candidates for parallel execution without speculation. Rather than searching the whole PDG for candidates for parallel execution, the scope of parallelization is localized to only control-equivalent children in this paper. There are two ways to relax this restriction: regions represented by control-equivalent siblings may be broken up into smaller regions in order to expose more parallelism, and regions that are descendants of sibling nodes may be promoted to be sibling nodes and speculatively executed before their conditions for execution are known. These generalizations are discussed in Section 7.

To keep the heuristic simple and fast, trade-offs are localized wherever possible. The formation of a partition depends only on local information: the latency of each child for all possible widths, dependence latencies among children, and synchronization overhead. In order to investigate how well each region may be parallelized across different numbers of processors, partitions are computed for every possible width. The possible widths are limited to powers of two in order to reduce the number of possible partitions per node. This set of *potential partitions* is used to provide estimates of that region's latency when forming the partition of that node's parent. Thus all of the trade-offs involved in tailoring the allocation of processors to available parallelism are made using information that is local to a node and its children.

4.3. Partitioning and Scheduling Procedure

The partitioning and scheduling procedure creates a partition of the entire PDG by specifying how each set of control-equivalent children is partitioned. In order to form a partition for a set of control-equivalent children, three things are needed: the estimated latency for each child at every possible width, estimated dependence latencies among all children, and estimated synchronization overhead required to maintain dependences if children are scheduled on different processors. The formation of a partition of a parent's children thus requires that the potential partitions have been computed for all of the child nodes first. This suggests a bottom-up traversal.

The partitioning and scheduling procedure has two phases: 1) forming potential partitions and estimating their latencies bottom-up, from the leaves to the root, and then 2) selecting partitions top-down, from the root to the leaves. The procedure is summarized in Figure 5. The first phase traverses PDG nodes in a depth-first fashion, determining the latencies of child nodes in order to form potential partitions among siblings.

Latency estimation begins at the leaf nodes, i.e. Code and Call nodes. The operations associated with Code nodes are executed on one superscalar processor. The latency of a Code node is determined by fine-grained scheduling its operations. The latency of a Call node, i.e. the latency of the called procedure, is determined from interprocedural information which is either entered by the user or automatically generated by PEDIGREE.

Next, a set of potential partitions is formed for non-leaf nodes, one potential partition for each width, i.e. for *1,2,4,..,Max* processors. The latencies of child nodes for each possible width have already been computed, since the traversal is bottom-up. The dependence latencies among those children are computed, and estimates of synchronization overhead for overlapping regions are made. PEDIGREE can use profiling information and feedback to deal with branch probabilities, variable latencies, and other uncertainties in this estimation process. Those latencies are then used when forming the potential partitions at the next level up. This bottom-up formation of potential partitions and propagation of their latencies proceeds up to the root of the PDG.

At the end of the bottom-up phase, all of the information necessary to make a global decision is available. The best partitions for every width for every node have been computed. The partition of the Proc node is selected, and the selection of partition widths is propagated down the PDG. The total number of processors that the procedure executes on is known. A partition has already been formed for that maximum width, and that partition specifies the processors that each child of the Proc node should execute on. Specifying those processors determines which potential partition should be used for each child. This process continues top down to the leaves of the PDG.

The challenging part of this process is forming each of the potential partitions. The key features of that process are highlighted here. Forming a partition is equivalent to scheduling the control-equivalent children. This scheduling process must order the children, determine their widths, and assign them to a subset of the processors. A typical list scheduling approach is used. First, a data flow graph is formed, where nodes in the graph represent control-equivalent regions, and weighted arcs represent data dependences between them. Then this graph is list scheduled. Early and late times are computed so that nodes with the minimum slack [SKAH91] are given highest priority. Scheduling proceeds forward in time. At any given cycle, the highest priority node is selected from among those that are data ready. For this node, the best width is determined, based on its latency for different widths (propagated in the bottom-up traversal), whether it is on a critical path, how much it would delay other nodes if it were widened, how synchronization overhead would be affected, and based on characteristics of the graph such as its parallelism profile.

Consider a program that has two loop nests. The first loop nest, L2, has a recurrence. The second loop nest, L4, reads an array that is written by the first loop nest, but is readily parallelizable. Suppose that L2 takes 10,000 cycles to execute on one processor, and 9,000 cycles on two processors. Let L4 take 20,000 and 12,000 cycles for one and two processors, respectively. Figure 6a shows a portion of the PDG for this program. Children of the Proc node are annotated with the latencies of partitions for widths 1 and 2, as computed earlier in the bottom-up traversal. No latencies are given for Code nodes for two processors, since Code nodes must execute on a single processor. Figure 6b shows the DFG formed by the children of the Proc node. C1 is scheduled first, because it has no data dependence predecessors and is on the longest dependence chain. It is assigned to processor 1. C3 is the next data-ready node, and it is scheduled on processor 2. L2 can begin execution before C1 completes. A choice must now be made: should L2 begin execution in cycle 10 on processor 2, or should it start in cycle 15 and be spread across both processors? If L2 uses both processors, it takes away resources from other nodes. It will also lengthen the critical path through the DFG, so it is assigned to processor 2. Next, a choice must be made about whether to spread L4 across both processors, since it can clearly benefit from using two processors. In that case it would start in cycle 10,010 and complete in cycle 22,010. Or, it could start executing in cycle 110 on one processor and complete in cycle 20,110. The latter choice leads to the best overall latency. The partition formed for this schedule is given in Figure 6c, and the record of execution is shown in Figure 6d. If the parent of C1, L2, C3 and L4 were something other than the Proc node, the latency would also be computed for executing those

Phase 1: Bottom-up Traversal: For each set of control-equivalent children, form a potential partition for every possible width, and estimate its latency

- *Analysis to be performed before forming a partition:*
  - estimated latency of each child, for every possible width, is already known - propagated up the PDG
  - estimate dependence latencies among children
  - estimate overhead of sync required to maintain dependences if scheduled on different processors
- *Form the partition:*
  - form a data flow graph with the children as nodes, dependence arcs weighted by dependence latency
  - list schedule the graph
    - prioritize according to dependence chain length
    - for highest priority node, considering sync overhead and impact on overall latency:
      - find best width
      - assign it to a processor subset

Phase 2: Top-down Traversal: Select the potential partition for each node that minimizes overall latency

- Use the number of processors specified in the machine description to select the width of the partition for the Proc node
- That partition specifies the width of each child, which in turn selects the partition to use for each child
- Recursively propagate the selection of partitions down to the leaf nodes

Figure 5. Partitioning And Scheduling Heuristic

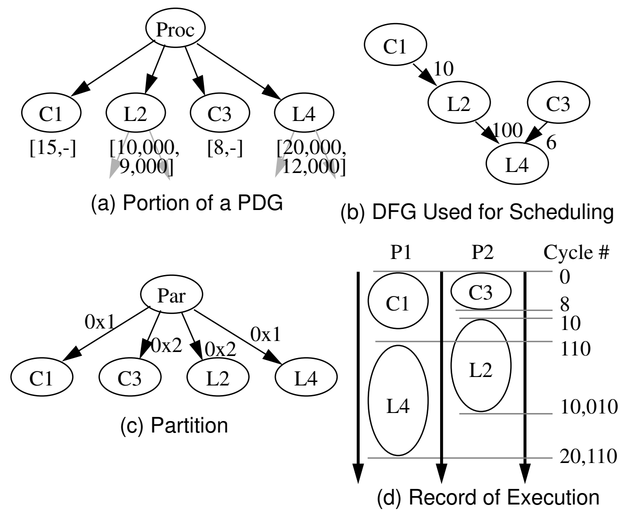

Figure 6. Partitioning and Scheduling Example

regions on only one processor. Since the procedure is executed on two processors, the partition of Figure 6c is selected, which mandates that the partition of L2 and L4 for only one node is selected.

The PDG readily exposes control parallelism in a hierarchical way, enabling parallelism to be exploited across a range of granularities. The bottom-up propagation of local information and top-down traversal that makes global decisions is clean and efficient. The traditional list scheduling approach is modified to allow nodes to be executed at various widths to help tailor the number of processors assigned to a region to its available coarse-grained parallelism. The approach taken here is to schedule code and assign it to subsets of processors statically, taking into account expected latencies and synchronization delays, and to let synchronization and control flow be resolved at run time.

5. Results

To demonstrate the effectiveness of this partitioning and scheduling approach we show the following: (1) exploiting coarse-grained parallelism on multiple processors yields better performance than exploiting only fine-grained parallelism within a single processor, (2) this approach significantly outperforms existing automatic partitioning and scheduling methods which parallelize only loop iterations, and (3) the partitioned schedule is tolerant of a reasonable amount of synchronization latency. Subsection 5.1 describes the experimental procedure used, and Subsection 5.2 describes the benchmark set. Presentation of performance data begins in Subsection 5.3, which covers the exploitation of coarse-grained parallelism. Subsection 5.4 explores the trade-offs involving simultaneously exploiting fine- and coarse-grained parallelism. Subsection 5.5 shows the impact of synchronization overhead on performance.

5.1. Experimental Procedure

All of the benchmarks used are written in C. The first step in compiling the benchmarks is to generate assembly code with `gcc -g -O -S`. The Alpha architecture is used as a platform for running these experiments. PEDIGREE compiles for several different target machines that vary in the number of processors, *N*, and the number of issue slots in each superscalar processor, *I*. Specific processor configurations are denoted by the expression *N* x *I*. For example, the sequence of configurations 2x2, 4x2, 8x2 represents SMPs of sizes 2, 4 and 8, in which each processor can issue two instructions per cycle. These configurations support an increasing amount of coarse-grained parallelism. The configurations 1x2, 1x4, 1x8, 1x16 represent single superscalar processors of increasing issue bandwidth. These configurations support an increasing amount of fine-grained parallelism. The target processor configuration that PEDIGREE compiles for is specified in the machine description files, in this case the 21064 implementation of the Alpha AXP ISA. It is a dual-issue, in-order superscalar. The functional units and memory ports are scaled linearly with the issue slots. While scaling the functional unit mix sublinearly would be more realistic, it might cloud the trade-offs between exploiting fine- and medium-grained parallelism highlighted in Subsection 5.4.

PEDIGREE emits code for each processor in the SMP, complete with synchronization primitives. This code is simulated with the PEDIGREE simulator to collect timing data and to verify correctness. PEDIGREE compiles most of the benchmarks used in this paper in a few seconds on an AlphaServer 8400 5/300. For very large procedures produced by inlining, compilation may take several minutes.

5.2. Benchmarks

Experimental data has been collected for the SDIO Signal and Data Processing Benchmark Suite, which consists of on-board signal processing kernels used in military and aerospace sensor system designs. The benchmarks consist of kernels that represent the compute-intensive portions of these applications. Most kernels begin with an initializing loop nest which is designed not to be parallelizable. This feature may be used to simulate a real-time stream of incoming data that is pipelined into each stage. The term *pipelined* is used to denote the overlap of stages of computation, where an earlier stage passes data to a later stage, and the later stage can begin execution before the earlier stage has completed. These stages correspond to processors in the SMP. An application is *pipelinable* if this type of coarse-grained control parallelism can be exploited. Most kernels consist of a set of loop nests and library calls to math functions which can be overlapped. Some of these loops are readily parallelizable, some are not. The structure of each kernel is briefly described in Table 1.

5.3. Exploiting Coarse-Grained Parallelism

The first experiment investigates the amount of coarse-grained parallelism available in these benchmarks when executed on an SMP, and compares this to exploiting exclusively data parallelism. Table 2 shows the increase in performance as the number of processors each with two

Table 1: Structural Description of the Kernels from the Strategic Defense Initiative Organization Benchmarks

Benchmark	Description
Area Centroid (ac)	init nest, non-parallelizable; recurrence loop on previous array
Aimpoint Offset (aim)	sequence of trig functions
Background Subtraction (back)	big init nest, non-parallelizable; small parallelizable init nest; doall loop on input arrays; small checksum loop
Object Classification (class)	init nest, non-parallelizable; recurrence loop on the initialization array; recurrence loop on the initialization array; doall loop on previous array; small checksum loop
2-D Convolution (conv)	init nest, irregular nest, non-parallelizable; init nest, non-parallelizable; large 3-deep nest, innermost has a recurrence, outer loops parallelizable; small checksum loop
Hardbody Detection (detect)	init nest, non-parallelizable; doall nest on previous array; doall nest on previous array; recurrence nest on previous array
Gamma Spike Circumvention (gamma)	init 5-nest, non-parallelizable; recurrent 3-nest on previous array, requires loop interchange
Kalman Filter (kalman)	sequence of many loops, most of which are recurrent; a huge basic block; sequence of many loops, most of which are recurrent
Munkres Frame Correlation (munk)	a sequence of overlappable nests; a sequence of pointer-intensive nests
Orbit Prediction (orbit)	a huge basic block; small checksum loop
Pulse Matching (pulse)	init 2-nest, non-parallelizable; large outer loop encloses a sequence of triangular nests whose data access patterns do not permit much pipelining
Scene Stabilization (scene)	init nest, non-parallelizable; doall nest on previous array, indirect references; small checksum loop
Time Delay Integration (tdi)	two small parallelizable init nests; small init nest, non-parallelizable recurrence nest on previous array, multiple loops at the same level with unknown index small checksum loop
Transversal Filtering (trans)	two init nests, non-parallelizable; recurrence nest on previous array; small checksum loop

issue slots is increased. The numbers in the 1x2 column give the cycle count for executing each kernel on the base machine configuration. The numbers in the other columns are speedups with respect to this base configuration. A detailed analysis of the characteristics of each kernel and what kind of parallelism is exploited is deferred until the next subsection. The harmonic mean of the speedups for two, four, and eight processors are 1.32, 1.43 and 1.46, respectively.

Many of the loop nests in these benchmarks have recurrences which prevent them from being parallelized easily. The primary means of achieving speedups in these kernels is to overlap these serial loop nests by pipelining them across the SMP, i.e. executing them on different processors and passing data from one loop nest to the next as it is generated. PEDIGREE achieves such pipelining speedups of up to 1.84x on two processors. *trans* is the best example of the exploitation of predominantly data parallelism, achieving speedups of 1.80x, 2.41x and 2.87x on two, four, and eight processors.

Table 2: Exploiting Coarse-Grained and Data Parallelism

Benchmark	1 x 2 cycle ct	2 x 2 speedup	4 x 2 speedup	8 x 2 speedup
ac	40842	1.84	1.85	1.85
aim	449	1.09	1.09	1.09
back	72057	1.73	1.73	1.73
class	2906	1.13	1.39	1.39
conv	169638	1.32	1.94	1.95
detect	135265	1.52	1.68	1.78
gamma	142234	1.36	1.40	1.40
kalman	92295	1.80	2.39	2.61
munk	10365	1.01	1.01	1.08
orbit	9894	1.05	1.05	1.05
pulse	282177	1.53	1.57	1.59
scene	362462	1.07	1.10	1.11
tdi	22524	1.10	1.10	1.10
trans	152076	1.80	2.41	2.87
Harmonic mean of speedups		1.32	1.43	1.46

Table 3: Exploiting Exclusively Data Parallelism

Benchmark	2 x 2 speedup	4 x 2 speedup	8 x 2 speedup
ac	1.21	1.21	1.21
aim	1.00	1.00	1.00
back	1.24	1.44	1.57
class	1.00	1.00	1.00
conv	1.32	1.89	1.89
detect	1.13	1.22	1.27
gamma	1.23	1.24	1.22
kalman	1.80	2.39	2.39
munk	1.00	1.00	1.00
orbit	1.05	1.05	1.05
pulse	1.22	1.22	1.22
scene	1.07	1.10	1.11
tdi	1.04	1.07	1.08
trans	1.80	2.41	2.87
Harmonic mean of speedups	1.18	1.26	1.28

The parallelization techniques that are in use today do not exploit control parallelism by pipelining loops, and exploit only data parallelism among loop iterations. Table 3 shows the speedups across different 2-issue slot processor configurations when only data parallelism is exploited. For the 8x2 processor configuration, the harmonic mean of the speedups when exploiting both data and control parallelism is 1.46, and 1.28 when exclusively exploiting data parallelism. This roughly fifteen percent improvement demonstrates PEDIGREE's ability to exploit more general kinds of parallelism than existing methods.

5.4. Balancing Fine & Coarse-Grained Parallelism

Recall that there are two different kinds of parallelism to be exploited in an SMP made up of superscalar processors: fine-grained within each processor and coarse-grained across multiple processors. Since programs vary in the amount of each parallelism that they exhibit, the best processor configuration may vary by application.

Figure 7 presents the relative execution times for each kernel for ten configurations that vary in the amount of coarse-grained parallelism (*N* processors) and the amount of fine-grained parallelism (*I* issue slots) they support. The product of the number of processors and number of issue slots in each processor, *N* x *I*, is limited to a maximum of sixteen in this study. The z-axis measures the execution time relative to the base 1x2 configuration, which is in the far corner of each 3D bar graph. Execution times tend to drop off to the left as more issue slots are available for exploiting fine-grained parallelism, and to the right as more processors are employed. The inter-processor synchronization latency assumed for Figure 7 and Table 4 is two cycles.

For each benchmark, there is one *N* x *I* configuration which best supports the available parallelism in the benchmark. Adding more processors or issue slots to that configuration does not improve performance significantly. This configuration is called the *best configuration*. Table 4 summarizes the speedups for *N* = 2 and 4, and gives the "best" *I* for each benchmark. It also cites the best configuration overall. The best *I* and *N* configuration varies with the available fine- and coarse-grained parallelism, as seen from Figure 7. As would be expected, allowing *I* to increase beyond 2 to suit the available fine-grained parallelism boosts performance, yielding a harmonic mean of 1.56x and 1.61x for two and four processors, respectively, rather than the 1.32x and 1.43x given above for the 2x2 and 4x2 configurations. Note that the $N \text{ x } I \leq 16$ restriction enables better performance for a configuration with fewer processors when there is an abundance of fine-grained parallelism.

The amount and type of parallelism exploited depends on several factors: the number and relative latencies of loop nests, whether the access patterns in each loop nest can be determined, and whether those patterns are regular and consistent for each pair of loop nests. The SDIO benchmarks cover a range of these conditions. An analysis of the structure of the code (Table 1) and the resulting speedups for each of these benchmarks (Figure 7) provides insight into different partitioning scenarios.

Figure 7 illustrates the sharp drop in execution time for *orbit* in the transition from two to four issue slots, and very small drop in the transition from one to two processors. *orbit* has almost no control parallelism, and just enough fine-grained parallelism for four issue slots. In contrast, *kalman*'s abundant control parallelism among many loops and *trans*' data parallelism among loop iterations is supported effectively with additional processors. *back* and *gamma* have two main loops which can be pipelined. They are close enough in size to keep utilization high, unlike *ac* and *class*, but irregular enough to allow smaller loops to be overlapped without using more processors. *aim* benefits from being able to overlap the execution of some trig functions, but since the latency of library functions is not measured, the performance improvement is not as evident as it would be if the library code were accounted for by the simulator. PEDIGREE effectively chooses to overlap parallelizable loop nests with serial loop nests in *detect* because this leads to greater speedup.

Memory disambiguation is a key factor in parallelization. *munk* has some overlappable loops at the start, but the main sequences of loop nests involve pointer chasing. Not being able to disambiguate those pointers prevents much parallelization. *pulse* contains a series of pipelinable loop nests, but irregular communication patterns among triangular loops limit the number of processors that can be usefully exploited. *scene*'s second loop nest uses computed array indices, such that the communication patterns are not predictable. It is neither parallelizable nor pipelinable. *tdi* has an index to several arrays which is calculated with the `mod` function. This causes memory disambiguation to fail and prevents parallelization.

Several kernels benefit from more issue bandwidth within each processor. This is indicated by the way the relative execution time drops off to the left for the *aim*, *back*, *orbit* and *trans* kernels. In most cases, four issue slots is sufficient. The best configuration varies across

Table 4: Performance Summary Across Different Configurations

Benchmark	Best Speedup for 2 Processors (*i*)	Best Speedup for 4 processors (*I*)	Best Configuration (*N* x *I*)
ac	2.00 (8)	1.95 (4)	2x8
aim	1.34 (8)	1.24 (4)	2x8
back	2.03 (8)	1.87 (4)	2x8
class	1.27 (8)	1.47 (4)	4x4
conv	1.70 (8)	2.34 (4)	4x4
detect	2.11 (8)	2.09 (4)	4x4
gamma	1.47 (8)	1.46 (4)	4x4
kalman	1.96 (4)	2.39 (2)	8x2
munk	1.10 (8)	1.08 (4)	1x8
orbit	1.63 (8)	1.46 (4)	2x8
pulse	1.59 (4)	1.57 (2)	4x2
scene	1.32 (4)	1.34 (4)	2x4
tdi	1.28 (8)	1.25 (4)	2x8
trans	1.94 (4)	2.60 (4)	8x2
Harm. mean	1.56	1.61	

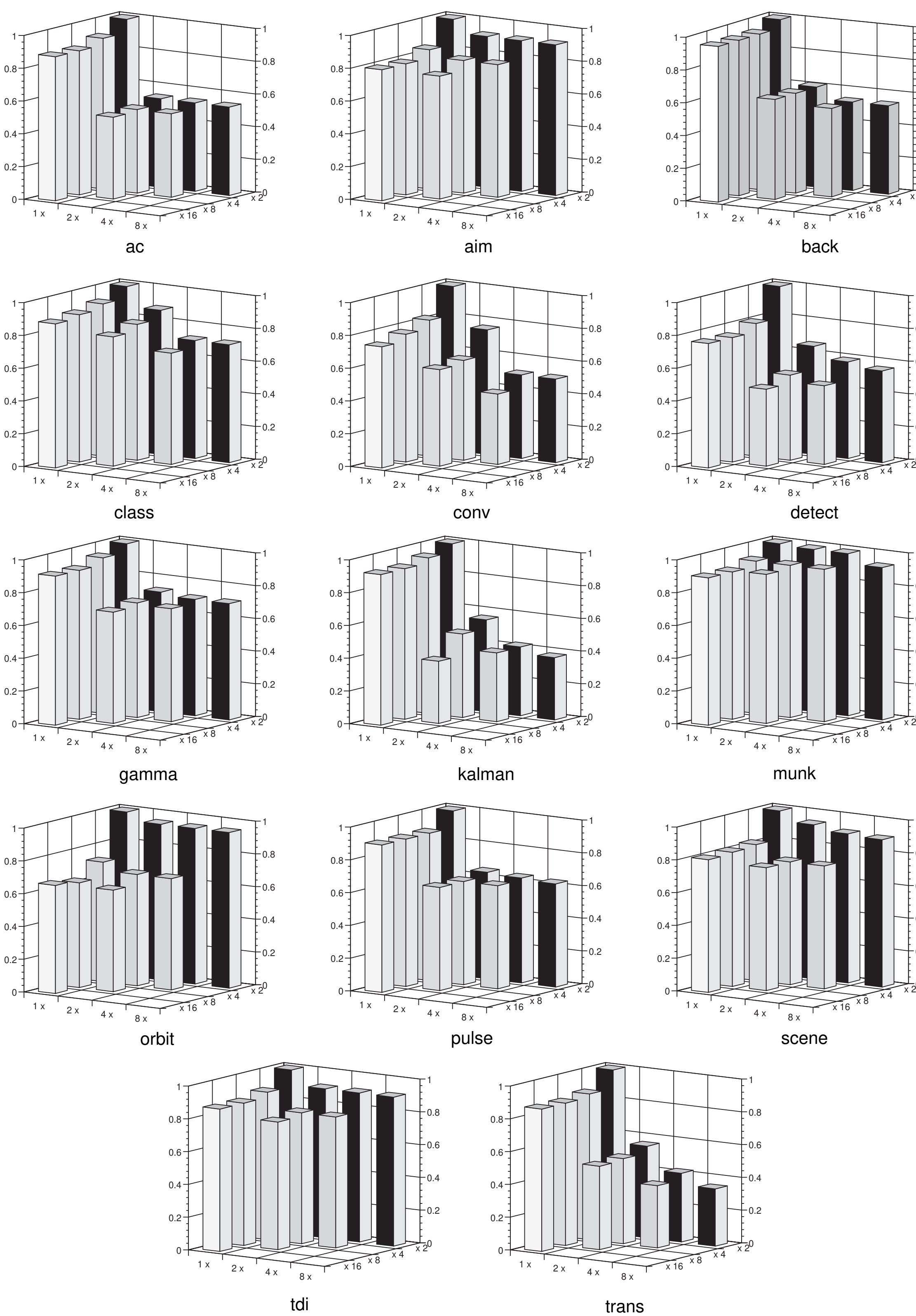

Figure 7. Relative Performance for Different N x I Configurations. N x I indicates the number of processors (N) and number of issue slots (I) in the machine configuration. The rear corner is 1x2. Support for fine-grained parallelism increases as issue slots are added (to the left from the rear corner), and support for coarser-grained parallelism increases as processors are added (to the right from the rear corner).

Table 5: Synchronization Overhead (%) With Respect to 2 Cycle Latency

Benchmark	2	5	10	20	50	100
ac	0.00	4.68	4.84	5.24	5.80	6.06
aim	0.00	0.00	7.78	15.57	19.70	27.98
back	0.00	0.00	0.01	0.04	0.11	0.23
class	0.00	0.76	1.62	4.41	8.00	10.16
conv	0.00	12.10	20.03	26.27	81.26	91.74
detect	0.00	28.37	28.38	28.39	28.46	28.46
gamma	0.00	0.63	0.86	0.34	0.51	0.53
kalman	0.00	0.03	0.09	0.20	0.26	0.29
munk	0.00	0.31	0.52	1.35	2.75	5.04
orbit	0.00	0.68	1.87	4.10	10.23	21.89
pulse	0.00	0.35	0.39	0.61	0.72	1.12
scene	0.00	1.03	-0.19	1.08	-1.37	-3.87
tdi	0.00	0.27	0.37	0.44	0.62	1.35
trans	0.00	0.00	0.01	0.03	0.09	0.20

benchmarks, indicating that tailoring the balance of fine-grained parallelism within each processor and coarse-grained parallelism across multiple processors is useful.

5.5. Sychronization Overhead

Synchronization overhead can play a big role in the types of parallelism that can be exploited. The synchronization overhead incurred by excessive bidirectional inter-processor communication can easily undo the benefits gained by parallelization. However, when computation can be pipelined in a coarse-grained data flow fashion between processors, the inter-processor dependences are mostly unidirectional, and significant synchronization latencies can be tolerated. For this subsection, all benchmarks are compiled with a specification of the sync delay. Table 5 shows the percent increase in execution time for machine models with up to one hundred cycles of synchronization latency relative to the base case of a two cycle synchronization latency. Data is given for the 2x8 processor configuration.

Benchmarks with pipelined and parallelizable regions, like *gamma* and *kalman* are very tolerant of latency. Parallelized doacross loops such as in *conv* and code with small granularity regions relative to the number of dependences, such as *aim*, are strongly affected by synchronization latency. A higher synchronization cost may prevent loop parallelization, such as in *ac*, or even lead to better performance with a more conservative schedule, as for *scene*. Even for a latency of 100 cycles, the mean overhead is only 10%. It is typically less than 1% for lower values.

6. Conclusions

This paper describes a compiler that exploits control and data parallelism across a range of granularities within a unified framework, doing so without language assistance or even the availability of the high-level language source code. The PEDIGREE compiler automatically partitions an assembly program based on its hierarchical control dependence structure, using the program dependence graph. The target machines for this paper are SMPs that support the exploitation of fine-grained parallelism within each processor and coarse-grained parallelism across multiple processors. PEDIGREE exploits both of these forms of parallelism. Each region of the program such as a loop nest or conditional can be spread across multiple processors if it contains coarse-grained parallelism, and each such region can be overlapped with other regions in order to minimize the program's overall execution latency.

The SDIO Signal and Data Processing Benchmark Suite has been parallelized using the partitioning and scheduling techniques introduced in this paper. PEDIGREE successfully exploits coarse-grained parallelism across multiple processors. For two processors with four issue slots each, speedups with respect to a single two-issue superscalar of as much as 1.95x are obtained through coarse-grained pipelining, and up to 2.60x by exploiting data parallelism. Seven of the benchmarks achieve speedups of over 50%, at least in part by exploiting pipelining parallelism. Only two of the benchmarks achieve speedups of over 50% from data parallelism alone. In general, the ability to exploit both control parallelism and data parallelism by overlapping regions, pipelining loop nests, and parallelizing loops where appropriate, produces better results than exploiting data parallelism alone. Exploiting only data parallelism within nested loops yields an average (harmonic mean) speedup of 1.28x for the 8x2 processor configuration, as opposed to 1.46x when both data and control parallelism are exploited for the same configuration.

When loops are coarse-grain pipelined, they tolerate synchronization latency well, often better than a doacross loop, e.g. *conv*. For most benchmarks, the additional synchronization overhead is less than 6% as synchronization latencies are scaled from two cycles to one hundred, and less than 2% for lower latencies. This suggests that these techniques can be effectively leveraged to compile these types of programs for realistic SMP systems.

This paper presents early results of automatic parallelization across a range of granularities for on-board signal processing applications. The performance improvements produced by PEDIGREE for an SMP target architecture suggest that these techniques may provide an effective solution to compiling on-board signal processing applications for commercial SMP systems.

7. Future Work

The partitioning and scheduling heuristics presented here are currently being extended to yield better performance. Section 4 mentioned that the scope of parallelization is currently restricted to only control-equivalent children. This restriction may be relaxed in two ways. One way is to break up the regions represented by control-equivalent siblings into smaller regions using *loop fission*, *conditional fission*, or *loop peeling*. In loop fission, a loop is divided into two or more similar smaller loops, each with a loop body that is a portion of the original loop body. This may be done to make smaller loops that are more parallelizable. In conditional fission, a conditional is divided into two or more similar smaller conditionals, each with the same conditional branch but with only a portion of the original body of the conditional. For example, in Figure 3, conditional splitting might be used to break Code node C7 into two halves, C7a and C7b, each with its own enclosing conditional, P6a and P6b. Then P6a and P6b would be siblings in the PDG, with children P7a and P7b, respectively. The motivation for this might be to break off a dependent operation such as a store in order to allow the rest of the conditional to be overlapped with code that lexically precedes it. This is vaguely similar to the break transformation of [AJLS92]. Iterations of a loop can also be peeled off and overlapped with other regions, while the remainder of the loop is parallelized across multiple processors. This transformation creates the opportunity for exploiting different kinds of parallelism: control parallelism among the peeled iterations and other regions, and data parallelism within the remaining parallelized loop. In Figure 4, loop L4 is overlapped with loop L2 to provide the shortest latency. If L4 could use all of the processors after L2 completes, the overall latency would be decreased. This could be done by peeling off just enough iterations to overlap with L2, and parallelizing the rest of the loop across both processors.

The second way to relax the restriction of considering only control-equivalent siblings for parallel execution is to promote regions that are lower in the hierarchy to be executed in parallel with their ancestor's siblings. Executing a region before its conditions for execution have been determined is a form of speculative execution. Speculation can be done in a controlled manner using the PDG, since it makes actual control dependences explicit. These two methods of exposing and exploiting more parallelism will be investigated in the future.

Another optimization is to be more aggressive in selecting the best schedule for a set of control-equivalent nodes. The optimal branch and bound techniques of [KP96] that have been applied to data partitioning on a multiprocessor may prove to be very useful for this purpose.

8. References

[AALL93] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An Overview of a Compiler for Scalable Parallel Machines. In *6th Workshop on Lang. and Compilers for Par. Comp.*, August 93.

[ABC+88] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A Framework for Determining Useful Parallelism. *ICS*, 1988.

[AJLS92] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced Region Scheduling on a Program Dependence Graph. In *MICRO 25*, 72–80, 1992.

[Ari82] M. Ben Ari. *Principles of Concurrent Programming*. Prentice Hall, 1982.

[Arn82] C. N. Arnold. Performance Evaluation of Three Automatic Vectorizer Packages. *ICPP*, 235–242, 1982.

[BCG+94] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *The First International Workshop on Parallel Processing*, 322–330, Bangalore, India, December 1994.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, August 1992.

[Cyt87] R. Cytron. Limited Processor Scheduling of Doacross Loops. In *ICPP*, 226–234, August 1987.

[ERA91] H. El-Rewini and H. H. Ali. Scheduling Task Graphs Containing Branches on Parallel Systems. In *HICSS*, 448–57, 1991.

[GBD+91] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *A Users' Guide to PVM Parallel Virtual Machine*. Oak Ridge National Laboratory, July 1991.

[GOS94] T. Gross, D. O'Hallaron, and J. Subhlok. Task parallelism in a High Performance Fortran framework. *IEEE Parallel & Distributed Technology*, (3):16–26, 1994.

[GP88] M. Girkar and C. Polychronopoulos. Partitioning Programs for Parallel Execution. In *ICS*, 216–29, St. Malo, France, July 1988. ACM.

[GP94] M. Girkar and C. Polychronopoulos. The Hierarchical Task Graph as a Universal Intermediate Representation. *International Journal of Parallel Programming*, 22(5):519–551, October 1994.

[GY92] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Jour. of Par. and Dist. Computing*, 16(4):276–91, December 1992.

[Hig93] High Performance FORTRAN Forum. *High Performance FORTRAN Language Specification*, 1.0 edition, May 1993.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The Program Structure Tree: Computing Control Regions in Linear Time. In *PLDI*, 171–85, Orlando, FL, June 1994. ACM.

[KP96] W. Kelly and W. Pugh. Minimizing Communication while Preserving Parallelism. In *ICS*, 52–60. ACM, 1996.

[NHS93] Chris Newburn, Andrew Huang, and John Shen. Balancing Fine- and Medium-Grained Parallelism in Scheduling Loops for the XIMD Architecture. In *Proceedings of the Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 39–52, Orlando, Florida, January 1993. IFIP.

[Nic91] Nichols Research Corporation. *Software User's Manual for the SDIO Signal and Data Processing Benchmark Suite for Single Processors*, 1991.

[NNS94] C. Newburn, D. Noonburg, and J. Shen. A PDG-Based Tool and Its Use in Analyzing Program Control Dependences. In *PACT*, 157–68, Montreal, Canada, August 1994. IFIP.

[Ost87] A. Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Computer Systems, 1987.

[RNSB94] L. Roh, W. Najjar, B. Shankar, and A.P.W. Bohm. An Evaluation of Optimized Threaded Code Generation. In *PACT*, 37–46, Montreal, Canada, August 1994. IFIP.

[Sar91] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Jour. of Res. and Dev.*, 35(5):779–804, September 1991.

[SFG91] K. Schwan, H. Forbes, and A. Gheith. A C Thread Library for Multiprocessors. Technical report, Georgia Institute of Technology, 1991.

[SH86] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *Proceedings of the Conference on LISP and Functional Programming*, 202–211, 1986.

[SKAH91] M. Smotherman, S. Krishnamurthy, P.S. Aravind, and D. Hunnicutt. Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling. In *MICRO 24*, 93–102, November 1991.

[Thi92] Thinking Machines Corporation. *Technical Summary of the CM5*, 1992.

[THK93] C. Tseng, S. Hiranandani, and K. Kennedy. Preliminary experiences with the Fortran D compiler. In *Supercomputing '93*, 338–350, Portland, OR, November 1993.

[TRG+87] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young. Mach Threads and the Unix Kernel: The Battle for Control. In *USENIX*, 185–197. 1987.

[YFGS95] T. Yang, C. Fu, A. Gerasoulis, and V. Sarkar. Mapping Iterative Task Graphs on Distributed Memory Machines. In *ICPP*, 151–8, August 1995.