# MIN-CUT-BASED PROGRAM DECOMPOSITION FOR THREAD-LEVEL SPECULATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Troy A. Johnson

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2003

# ACKNOWLEDGMENTS

I would like to thank my advisors, Professors Eigenmann and Vijaykumar, for many helpful suggestions in completing this project. Many thanks also go to Chong Liang Ooi for his prompt attention to simulator bugs and to Professor Li for participating on my thesis committee. The members of the Paramount Research Group provided useful feedback and thought-provoking discussions concerning my work.

Completing a graduate degree is impossible without the support of friends and family. In particular, I want to acknowledge the members of HKN for letting me become involved with the organization at Purdue and participate in their activities, although I joined as part of a different chapter. Thanks to Craig and A.J. for giving me some friends in the grad house to hang out with, and to Kathy and Beth for a tremendous amount of support despite being far away.

# TABLE OF CONTENTS


Page

LIST OF TABLES . . . v

LIST OF FIGURES . . . vi

SYMBOLS . . . vii

ABBREVIATIONS . . . viii

ABSTRACT . . . ix

1 INTRODUCTION . . . 1

- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Speculative CMP Background . . . 3
  - 1.3.1 Architecture . . . 3
  - 1.3.2 Execution Model . . . 4

2 COMPILER ALGORITHM FOR THREAD DECOMPOSITION . . . 9

- 2.1 Goals and Rationale . . . 9
  - 2.1.1 Goals . . . 9
  - 2.1.2 Possible Approaches . . . 10
  - 2.1.3 Our Approach . . . 11
- 2.2 Algorithm Input . . . 12
- 2.3 The Algorithm . . . 12
  - 2.3.1 Overall Operation . . . 12
  - 2.3.2 RemoveLoops: *Handling Cycles in the CFG* . . . 13
  - 2.3.3 DecomposeThread: *Load-Balanced Min-Cut* . . . 14
  - 2.3.4 L: *Load Imbalance* . . . 16
  - 2.3.5 S: *Thread Size* . . . 16
  - 2.3.6 D: *Dependence Penalty* . . . 17

Page


2.3.7 P: *Prediction Penalty* . . . . . . . . . . . . . . . . . . . . . . . . 18

2.3.8 Profitable: *When to cut?* . . . . . . . . . . . . . . . . . . . . . . . 20

2.4 Compiler Implementation . . . . . . . . . . . . . . . . . . . . . . . . . 21

2.5 Obtaining Profile Information . . . . . . . . . . . . . . . . . . . . . . . 22

3 ANALYSIS OF SPEC CPU2000 . . . . . . . . . . . . . . . . . . . . . . . . 25

3.1 Simulator Configuration . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.3 Overhead Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.4 Algorithm Efficiency . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4 RELATED WORK . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

5 CONCLUSIONS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

LIST OF REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

## LIST OF TABLES


| Table | | Page |
|---|---|---|
| 2.1 | CFG File Format | 22 |
| 3.1 | Simulator Configuration | 26 |
| 3.2 | Baseline vs Improved Performance | 27 |
| 3.3 | Basic Blocks Per Procedure (BPP) and Elapsed Time for Heuristics vs Decomposition | 35 |

# LIST OF FIGURES


| Figure | | Page |
| --- | --- | --- |
| 1.1 | Rollback due to a data or control dependence violation. Thread numbers indicate sequential order. Thread 1 detects a violation in Thread 2. It rolls back thread 2 and the younger Thread 3, followed by the dispatch of new threads on P2 and P3. Threads 2b and 3b may or may not be the same as Threads 2 and 3. | 6 |
| 1.2 | Load imbalance. Strict cyclic dispatch order inherent in the architecture leads to load imbalance. | 7 |
| 2.1 | Individual vs simultaneous consideration of constraints resulting in different thread boundaries. CFG annotations show basic block sizes and branch probabilities. | 10 |
| 2.2 | Effect of cutting a dependence edge within a thread | 19 |
| 3.1 | SPEC FP2000 Improvement of our min-cut approach over the best known method | 29 |
| 3.2 | SPEC INT2000 Improvement of our min-cut approach over the best known method | 30 |
| 3.3 | SPEC FP2000 Relative importance of reduced overheads compared to 100% of original overhead | 32 |
| 3.4 | SPEC INT2000 Relative importance of reduced overheads compared to 100% of original overhead | 33 |

# SYMBOLS

| | |
|---|---|
| C | cut |
| $\mathcal{D}_i$ | dependence penalty for cutting edge $e_i$ |
| E | edge set |
| $e_i$ | a particular edge |
| G | graph {V, E} |
| $\mathcal{L}(C)$ | load imbalance created by C |
| $\mathcal{P}_i$ | prediction penalty for cutting edge $e_i$ |
| $\mathcal{S}(T)$ | size of thread T |
| T | thread |
| V | vertex set |
| $\mathcal{W}_i$ | $\mathcal{D}_i + \mathcal{P}_i$ |

# ABBREVIATIONS

| | |
|---|---|
| CFG | Control-Flow Graph |
| CMP | Chip Multiprocessor |
| SPEC | Standard Performance Evaluation Corporation |

## ABSTRACT

Johnson, Troy A. M.S.E.C.E., Purdue University, May, 2003. Min-Cut-Based Program Decomposition for Thread-Level Speculation. Major Professor: Rudolf Eigenmann.

With billion-transistor chips on the horizon, single-chip multiprocessors (CMPs) are likely to be commodity components within a few years. Speculative CMPs provide the compiler with the same interface as a standard, sequential processor by supporting the safe, simultaneous execution of potentially dependent threads. The compiler's additional responsibility is to decompose the sequential instruction stream into these speculatively parallel threads. For thread decomposition, the compiler faces multiple performance constraints related to data dependences, load imbalance, thread size, and thread prediction. We show that the speculative thread decomposition problem can be mapped onto and dealt with in terms of a balanced min-cut algorithm, allowing all constraints to be considered simultaneously. Our implementation of the algorithm uses profile information for data dependences and branch frequencies, along with basic block and function call cycle counts. Our method improves the (geometric) average speedup of floating-point programs by 13.8% and integer programs by 11.3% over the previous approach, which was based on several competing heuristics. Comparisons with manual thread selection suggest that our approximate solution to the NP-hard thread decomposition problem is close to optimal.

# 1. INTRODUCTION

## 1.1 Motivation

Single-chip multiprocessors (CMPs) are likely to be commodity components within a few years, as the number of transistors per chip will soon cross the one billion mark. CMPs may be operated as conventional multiprocessors, where parallelism is exploited explicitly from concurrently running applications, or from parallel sections within a single application. By contrast, speculative CMPs exploit parallelism implicitly from an application's sequential instruction stream. This paper deals with compiler issues for such speculative CMPs.

Speculative CMPs provide the compiler with the same interface as a standard, sequential processor while supporting the safe, simultaneous execution of potentially dependent threads. The compiler's additional responsibility is to decompose the sequential instruction stream into these speculatively parallel threads. The compiler defines *speculative threads* by inserting boundary marks into the sequential instruction stream to tell the CMP where to speculate; that is, which code segments to try to execute in parallel with each other. The CMP uses prediction to select and execute a set of threads while enforcing correctness, such that the program's output is consistent with that of its sequential execution. To enforce correctness, the CMP employs data dependence tracking mechanisms, keeps uncertain data in speculative storage, rolls back incorrect executions, and commits data to the memory system when speculative threads succeed. This speculative mechanism has been described formally from a software perspective in [1].

For *thread decomposition*, the compiler faces multiple performance constraints related to data dependence, load imbalance, thread size, and thread prediction. Ideally, no data dependence should cross a thread boundary to avoid dependence-violation

rollbacks; nearby threads should have equal size to avoid load imbalance; a thread should be large enough to amortize its dispatch and commit overhead, but small enough such that all of its speculative data can be buffered; and thread boundaries should be placed at predictable branches to avoid misprediction rollbacks.

A previous approach in [2] applied several different heuristics to build threads from basic blocks and loops. Load imbalance was not considered, and there was very limited dependence and prediction information. Basic blocks were grouped into threads using control-flow and data-dependence heuristics without any performance-based tradeoff between them. Thread boundaries conservatively appeared at the beginning and end of all loops, as well as at all but the smallest function calls, ignoring opportunities for coarser parallelism. In contrast to [2], this paper describes a more rigorous approach to thread decomposition, based on a balanced min-cut graph-partitioning algorithm. Our compiler algorithm simultaneously considers data dependences, thread sizes, and branch frequencies obtained from profile runs. It considers loops at all nesting levels and finds balanced, low-overhead decompositions of non-loop sections, which is crucial for non-numerical programs.

### 1.2 Contributions

Our main contributions are as follows:

- We are the first to show that the speculative thread decomposition problem can be mapped onto and dealt with in terms of a balanced min-cut algorithm.

- The balanced min-cut framework actively trades-off one constraint for another by considering all constraints simultaneously, unlike the previous approach which used several competing heuristics, each in isolation.

- We introduce a novel method for assigning edge weights such that the cost of cutting a control-flow edge approximates the number of cycles lost due to the cut.

- We have implemented the algorithm as part of a fully-automated profile-based compilation process, allowing us to measure 17 C and Fortran SPEC CPU2000 programs. Our method improves the (geometric) average speedup of floating-point programs by 13.8% and integer programs by 11.3% over the previous approach. Comparisons with manual thread selection suggest that our approximate solution to the NP-hard thread decomposition problem is close to optimal.

## 1.3 Speculative CMP Background

### 1.3.1 Architecture

Chip multiprocessors (CMPs) can be viewed as architectural alternatives to complex wide-issue processors for exploiting instruction-level parallelism. CMPs seek to group narrower processors together to achieve high performance. As clock speeds, wire delays, and pipeline stages increase, CMPs offer a simpler design than utilizing all chip space to build a single monolithic processor. Using smaller processors leaves room for additional hardware that can assist or even eliminate the need for advanced data-dependence analysis by parallelizing compilers [3,4]. *Speculative CMPs* allow the execution of multiple, possibly dependent threads. Hardware mechanisms detect if threads are misspeculated due to control or data dependence. *Speculative storage* (typically implemented through the cache) keeps uncertain data during speculation and commits changes when speculation completes. Main memory is non-speculative and appears the same to the user and the compiler as in a conventional single-processor system.

Several speculative CMP architectures have been proposed, including the Wisconsin Multiscalar [5], Minnesota Superthreaded [6], Stanford Hydra [7], MIT RAW [8], CMU STAMPede [9], and Illinois I-ACOMA [10]. The primary difference among them lies in the cache protocol they use for handling speculative storage and detecting misspeculation. Different cache protocols impact performance, but do not change the compiler's view of the execution model [11–15]. Thread-level speculation has also

appeared in virtual machines [16]. The issue of identifying speculative threads arises in all of these approaches.

### 1.3.2 Execution Model

At a high level, a compiler may view a speculative CMP as a machine that correctly executes sequential code, with compiler-inserted thread boundaries helping to optimize the execution. Alternatively, a compiler may view a speculative CMP as a traditional shared-memory multiprocessor, where the simultaneous execution of dependent threads does not result in incorrect program behavior, but instead only in a performance penalty. A more detailed execution model is necessary in order for a compiler to attempt to insert thread boundaries in the best possible way. We introduce the model in terms of the execution overheads that are affected by the compiler's choice of thread boundaries. For a thorough execution model from a compiler perspective, we refer to [1]. We assume the Multiscalar architecture, although our techniques also apply to other CMPs.

A thread dispatcher continuously picks threads from the sequential instruction stream and dispatches them to available processors. It uses prediction to decide which thread to dispatch next. A thread's execution may be incorrect either because the prediction was wrong, resulting in a control dependence violation, or because an interthread data dependence was violated. The CMP detects both types of violations and reacts by rolling back and restarting threads as necessary [11, 17]. The oldest thread in execution (w.r.t. sequential order) is always nonspeculative, guaranteeing progress, while all younger threads are speculative. A speculative thread keeps its modified data in speculative storage until it becomes the nonspeculative thread. Thread decomposition affects the following overheads.

### Data Dependence Rollback

True dependences that cross thread boundaries may lead to data dependence violations and cause rollbacks, as in Figure 1.1. A data dependence violation is detected at the write reference to a memory location that had previously been read by a younger thread. The reader and all younger threads are rolled back. The entire execution time of the rolled-back threads appears as overhead.

Only true memory dependences (read-after-write) cause violations. Anti (write-after-read) and output (write-after-write) dependences are properly handled in the CMP by the cache protocol and the speculative storage. Furthermore, the executable code will include register dependences between threads, especially if thread decomposition is done by the compiler after register allocation and other optimizations. Register dependences can be handled by the CMP [18]. Architectures have also evolved to dynamically synchronize any frequently-encountered memory dependences that impede parallel execution [19].

### Control Dependence Rollback

Control dependence rollbacks are caused by thread misprediction. A control dependence violation is detected when an older thread completes and its actual successor differs from the predicted successor. The overhead is the execution time of all the rolled-back, younger threads as in Figure 1.1.

### Load Imbalance

Threads of unequal size can lead to load imbalance, as in Figure 1.2. The imbalance stems from an architecture property: threads are dispatched to the processors in a cyclic order. Maintaining a cyclic dispatch order allows the sequence of threads to be easily determined for rollback operations. While this simplifies the architectural design, it increases load imbalance. If the processor that should receive the next

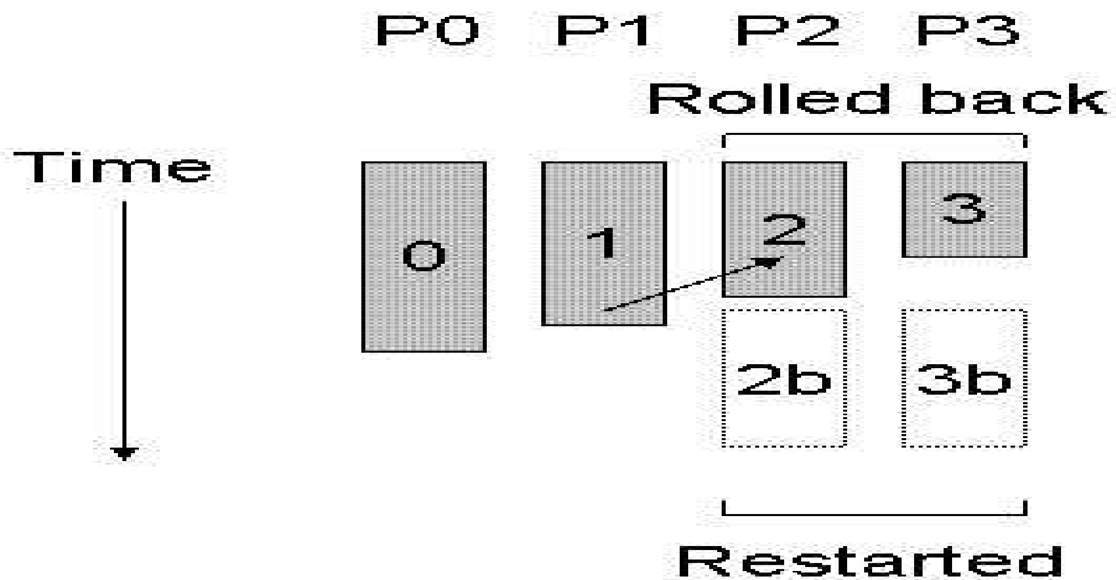


Fig. 1.1. Rollback due to a data or control dependence violation. Thread numbers indicate sequential order. Thread 1 detects a violation in Thread 2. It rolls back thread 2 and the younger Thread 3, followed by the dispatch of new threads on P2 and P3. Threads 2b and 3b may or may not be the same as Threads 2 and 3.

thread is still busy executing a previous thread, then the thread dispatch mechanism must wait before assigning a new thread. There may be other processors that are idle but do not satisfy the dispatch order. These idle cycles represent load imbalance.

**Size-Related Thread Overheads**

Although thread dispatch is efficient, dispatch and commit overhead overwhelms very small threads. Decomposing a program into large threads will reduce the significance of this overhead. However, large threads may overflow the speculative storage, because they will include more memory accesses. An overflow completely stalls speculative execution, until the oldest thread completes and frees some speculative storage. Typically, storage overflow only occurs in very large threads. In related work we have developed *memory reference idempotency* analysis techniques, which can reduce the

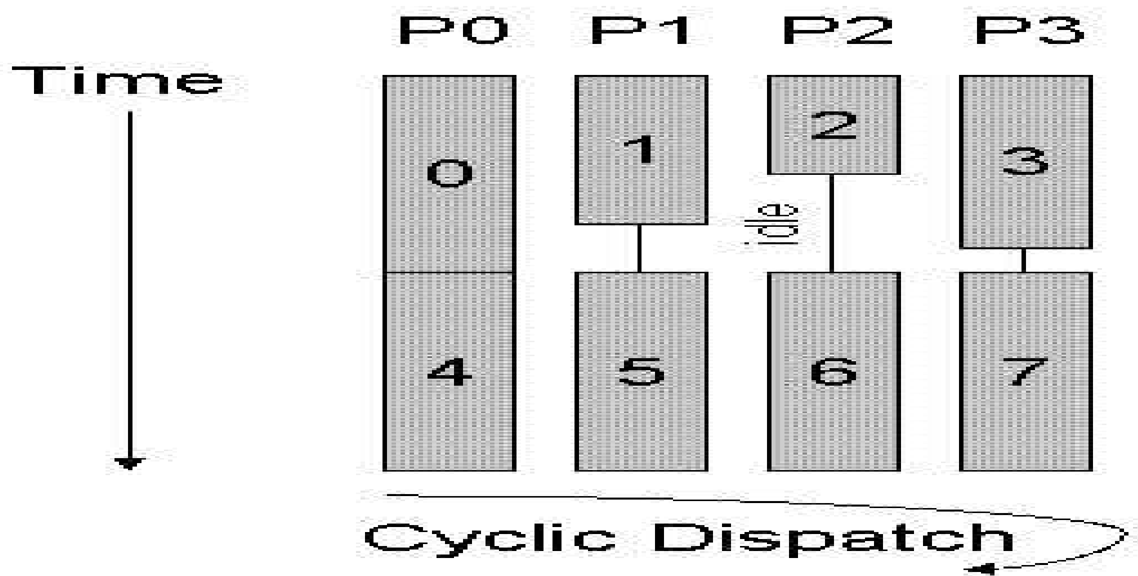


Fig. 1.2. Load imbalance. Strict cyclic dispatch order inherent in the architecture leads to load imbalance.

amount of speculative storage required by a program [1]. These techniques have not been applied to the experiments in this paper, however. An additional drawback of large threads is the potential increase of rollback overhead when a misspeculation occurs.

# 2. COMPILER ALGORITHM FOR THREAD DECOMPOSITION

## 2.1 Goals and Rationale

### 2.1.1 Goals

The goal of our algorithm is to decompose a program into threads to maximize the parallelism exposed to the underlying speculative CMP architecture, which in our case is Multiscalar. The compiler needs to carefully attend to each of the overheads introduced above, while placing thread boundaries on control-flow graph (CFG) edges. In our compiler implementation, thread boundaries are created after all other optimizations have been applied.

A thread begins at the first basic block of a procedure or after any thread boundary. The set of basic blocks in a thread is defined as the set of blocks reachable from its starting block without crossing a thread boundary. The compiler backend replicates blocks as necessary to package code into threads, so it is not necessary for the boundaries to create disjoint sets. Function calls are special: they can either be included or excluded from threads. If included, any thread boundaries within the called procedure are suppressed at run time; if excluded, thread dispatch will continue with the threads in the called procedure. Call suppression is our only means of making context-sensitive decisions for function calls, since thread boundaries within a procedure are the same no matter the location of the call. The decision to include or exclude a call is made at each call site on the CFG. Hence, a procedure can execute as part of the calling thread, or execute as multiple threads.

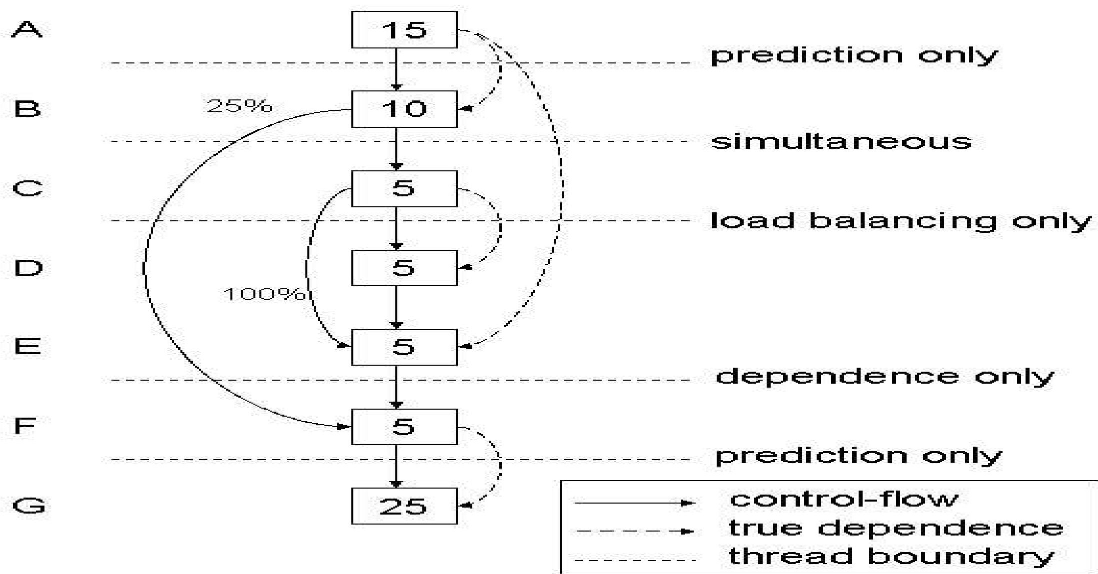


Fig. 2.1. Individual vs simultaneous consideration of constraints resulting in different thread boundaries. CFG annotations show basic block sizes and branch probabilities.

### 2.1.2 Possible Approaches

A possible approach to defining threads would be to apply techniques similar to those used in today's parallelizing compilers [3, 4]. These techniques focus on identifying loops with independent iterations. Such loops would yield independent threads, eliminating data-dependence rollbacks. However, we have found that such techniques would be insufficient for non-numerical programs, which tend to include short, irregular loops containing many dependences. Parallelism also needs to be extracted from non-loop code sections.

In early experiments, involving both loop and non-loop sections, we found that naïve thread-boundary placement benefits some programs, while increases overheads in others. Ignoring one of the sources of overhead described in Section 1.3.2 and focusing on the others results in shifting around the overheads with minimal improvement

to their total. A heuristic considering only prediction will tend to place boundaries where there is no control flow (i.e., near post-dominators), but ignores dependence and load imbalance. A purely load-balancing algorithm may place boundaries across any number of dependence or control-flow edges, greatly increasing the probability of rollbacks. A dependence-only heuristic will lead to unbalanced threads and place boundaries across any number of control-flow edges. Consequently, an algorithm that considers all constraints simultaneously may select different boundary locations than any of these three individual methods, as illustrated in Figure 2.1, and its solutions will generally incur less total overhead.

### 2.1.3 Our Approach

Our algorithm actively trades off one constraint for another, to reduce total overhead. Optimal thread decomposition that makes this tradeoff is not feasible, since balanced $k$-partitioning of a graph, even without additional tradeoffs, is known to be NP-hard [20]. Our approach is to use a *balanced min-cut* graph-partitioning algorithm to split the single-thread, sequential program into smaller threads. Large serial threads are cut if they can provide better parallelism in the form of smaller segments, as determined by trading-off the increased parallelism against the incurred overhead. This top-down approach allows us to reach a solution with a larger average thread size than if we built threads upward from basic blocks, as in [2].

In applying a polynomial-time, balanced min-cut algorithm to obtain a good partitioning, we use a technique that has proven useful in other areas. For example, in [21, 22] it was applied to circuit placement and routing. We add two important extensions to the basic algorithm used in [21] and, to our knowledge, are the first to apply it to a compiler optimization problem:

1. The cut-edge weights and the imbalance created by a cut are given equal consideration, so the algorithm terminates when it reaches a minimum combined cost instead of when imbalance falls below some threshold.

2. It was necessary to redefine the notion of a balanced bipartition, normally the size difference of two parts, to model load imbalance, which must account for multiple execution paths.

## 2.2 Algorithm Input

Our algorithm's input is an annotated CFG, where each vertex represents a basic block of the program. The annotations include static information about source line numbers, called functions, and instruction counts. They also include dynamic information about branch frequencies, average execution cycles per function, and dependences. Our profiling methods are described in Section 2.5.

## 2.3 The Algorithm

### 2.3.1 Overall Operation

We begin with a single-thread program, meaning that there are no thread boundaries and all function calls are included. The algorithm then inserts boundaries in each procedure. The first part of the algorithm deals with loops in the CFG and the second part uses the min-cut decomposition algorithm:

For each procedure $P \in$ Program
  RemoveLoops($P$)
  While ($\exists$ thread $T \in P$ s.t. $T$ is not marked indivisible)
    $C \leftarrow$ DecomposeThread($T$)
    if Profitable($C$, $T$)
      ApplyCut($C$)
    else
      $T \leftarrow$ indivisible

Each decomposition step seeks a low-cost, balanced cut of $T$'s CFG. We define the cost as the sum of the cut edge weights and the load imbalance among the created

threads. Weight assignment and load imbalance are explained in later sections, where we will continue our top-down description of the algorithm. The cut splits the graph into two partitions, where source vertices of cut edges become thread exits, and sink vertices of cut edges become thread entries. If cut $C$ is deemed profitable (i.e., if it is expected to perform better than not cutting), then ApplyCut inserts thread boundaries. Otherwise, $T$ is marked *indivisible* so that it will no longer be considered.

### 2.3.2 RemoveLoops: *Handling Cycles in the CFG*

RemoveLoops($P$) inspects cycles in the CFG of procedure $P$. Loops need special consideration because both our compiler's internal representation and the architecture dictate that thread boundaries placed in a loop affect all iterations. A solution that would conceptually fully unroll all loops in the program and then decompose the resulting single, large thread into appropriately-sized segments is therefore not feasible. This limitation is tempered by the fact that very small loops have already been unrolled in the course of normal optimization prior to applying our algorithm. CFG cycles are handled as follows:

**Fully parallel loops:** Loops without any cross-iteration dependences, as estimated by profiling, are decomposed into exactly one thread per iteration because the threads will likely be free of any crossing dependences in the experimental run. This is the overriding criterion since consecutive iterations are easily predicted and they will tend to be balanced because they contain the same code. Dependences with a distance greater than the number of processors are ignored, since they cannot affect the speculative execution. The iterations are marked indivisible so they will not be cut later.

If the speculative CMP also supports explicit threading [23], a compiler capable of testing for data dependences can mark provably parallel loops [3, 4]. Explicit threads can be executed more efficiently, because dependences need not be tracked and there

is no speculative state to maintain. This capability is not used in the present paper, however.

**Dependent loops:** All other loops, not included within a parallel loop, are decomposed into one thread per iteration. Such loops remain exposed to the algorithm for further decomposition, unlike parallel loops. An important consequence is that, for all loop regions considered by the cutting algorithm, the back-edges of the CFG are already thread boundaries, so there is no further special handling required for graph cycles.

**Dependent loops:** All other loops, not included within a parallel loop, are decomposed into one thread per iteration. Such loops remain exposed to the algorithm for further decomposition, unlike parallel loops. An important consequence is that, for all loop regions considered by the cutting algorithm, the back-edges of the CFG are already thread boundaries, so there is no further special handling required for graph cycles.

#### 2.3.3 DecomposeThread: *Load-Balanced Min-Cut*

DecomposeThread($T$) cuts $T$'s CFG into two parts, along the best cut $B$, creating two or more threads. $G$ is a CFG $\{V,\ E\}$ where $V$ is the set of all vertices (basic blocks) in the thread and $E$ is the set of edges connecting them. Let $s$ be $T$'s entry vertex and $t$ be an extra vertex attached to all thread exits. Initially $Cost(B) = \infty$.

1. Compute weights $\forall$ edges $e_i \in E$. The weight $\mathcal{W}_i$ of each edge $e_i$ is computed as the sum of its dependence and prediction penalties: $\mathcal{W}_i = \mathcal{D}_i + \mathcal{P}_i$. $\mathcal{D}$ and $\mathcal{P}$ are defined in later sections.

2. Compute a min-cut $C$ for $G$. We use the Edmonds-Karp variation of the Ford-Fulkerson method which has $O(VE^2)$ time complexity [24, 25]. Let $V_1$ be the set of vertices containing $s$ and $V_2$ be the set containing $t$.

3. Temporarily collapse all vertices in $V_1$ ($V_2$) to the $s$ ($t$) vertex if the smaller half is $V_1$ ($V_2$).

4. Iteratively try swapping all border vertices from the larger of $V_1$ and $V_2$ across $C$ and find a new min-cut for each. Additionally try to exclude from $T$ function calls bordering $C$. Let the new $C$ be the min-cut whose Cost was the least. $Cost(C) = \sum \mathcal{W}_i + \mathcal{L}(C)$, where $\mathcal{W}_i$ represents the weights of edges cut by $C$ and $\mathcal{L}(C)$ is the load imbalance created by $C$.

5. If $Cost(B) < Cost(C)$, stop and return $B$; Else $B = C$, go to Step 3.

The algorithm makes use of a well-known min-cut algorithm to find a cut of minimum cost. This first cut, in Step 2, is the overall best cut we can make if we ignore load imbalance. However, we are interested in all three constraints, so the next goal is to determine if there is another cut that may have a slightly higher $\sum \mathcal{W}_i$ but lower $\mathcal{L}(C)$, such that the expected number of cycles lost, represented by the $Cost(C)$, is reduced.

Steps 3 and 4 aim to gradually shift the current cut by moving nodes bordering the cut to the other side of the cut. All nodes on one side are temporarily collapsed to a single node, and a new min-cut is computed for each possible node that can be swapped across the cut. The collapse is necessary to prevent the min-cut algorithm from returning the initial cut every time. The new current cut is the best cut made by swapping nodes. In Step 5 we see if the new cut has improved the overall best. If not, the algorithm stops.

In our implementation, the balancing step does not repeatedly compute full min-cuts. Instead, the algorithm incrementally builds on the previous step. This efficiency measure was proposed by [21] and leads to an overall algorithm complexity of $O(VE^2)$. In the case of a CFG, where each vertex has at least one and at most two outgoing edges, $(V-1) \leq E \leq 2V$, so the operation is $O(V^3)$. In the context of our algorithm, we must set edge weights (which we will show to be $\Theta(V^2)$) and perform a balanced cut repeatedly for each procedure in the program. A very unlikely worst case is that

the balancing step repeats until every edge is cut, resulting in a $O(V^4)$ algorithm. In practice, our results show the run time of the algorithm to be very acceptable, as discussed later in Section 3.4.

### 2.3.4 L: *Load Imbalance*

$\mathcal{L}(C)$ represents the load imbalance of a thread's CFG, partitioned along the cut $C$. When cutting the CFG of a thread into two parts, new threads begin at the sinks of the cut edges. Therefore, the cut results in one thread followed by one or more successors. Let the size of the first partition be $\mathcal{S}(T_1)$ and the rest $\mathcal{S}(T_2)...\mathcal{S}(T_n)$. Using branch frequencies, we compute the weighted execution time $\mathcal{S}(T_{avg})$ of $\mathcal{S}(T_2)...\mathcal{S}(T_n)$. Thus, $\mathcal{L}(C) = |\mathcal{S}(T_1) - \mathcal{S}(T_{avg})|$.

The computed value is an estimate, because the relative starting times of the threads are unknown during compilation. It could be the case that a set of equally-sized threads would not be balanced at run time, if their starting times were staggered due to some previous load imbalance. We make the assumption that such previous load imbalance is not severe, given that we would have attempted to choose equally-sized threads there as well. Computing the load imbalance requires a size calculation for each thread, later shown to be $\Theta(V)$. However, the number of threads created by a cut is very small (typically between 2 and 5 in our experiments) so $\mathcal{L}(C)$ is a $\Theta(V)$ operation.

### 2.3.5 S: *Thread Size*

$\mathcal{S}(T)$ is the size of thread $T$ in terms of number of cycles. $T$ may contain several possible control paths. Hence, the dynamic size of the thread depends on the path taken. Using profiling information, we compute an expected value for the dynamic size of $T$:

$$\mathcal{S}(T) = \sum_{k=1}^{paths} p_k s_k \tag{2.1}$$

We represent the probability that path $k$ is taken by $p_k$ and the size of path $k$ as $s_k$. The probability of a path is the product of the branch frequencies along the path. The size of the path is computed as the sum of static instructions for basic blocks and dynamic cycles for function calls along the path, similar to estimating procedure execution time in [26]. However, we use the number of dynamic cycles for calls instead of static estimation for every procedure in the program, and we are interested in execution times for the various subsets of the CFG that are being considered for threads.

Computing an expected thread size is a common operation in our algorithm, so it is desirable to make it as efficient as possible. We recognize that equation (2.1) can be computed recursively, through a depth-first search of the CFG:

$$\mathcal{S}(T) = s_{init} + p_{true}\mathcal{S}_{succ}(true) + p_{false}\mathcal{S}_{succ}(false) \tag{2.2}$$

The expected thread size is the size of the initial block plus the size of its successor paths. The computation for nodes with zero or one successors is obvious; equation (2.2) gives the computation for the two successors of a branch. The two successor sizes are weighted by their probability. We optimize further by recognizing that, due to control-flow reconvergence, the expected size for a subpath may be needed multiple times. By *memoizing* the sizes of sub paths, we greatly reduce the amount of work necessary. The resulting algorithm is a standard depth-first search with $\Theta(V + E)$ complexity, or simply $\Theta(V)$ for our CFG.

### 2.3.6 D: *Dependence Penalty*

$\mathcal{D}_i$, the dependence penalty due to cutting edge $e_i$, is estimated from dependence profile information. If a thread boundary must be placed across a dependence, the likelihood of it causing a rollback generally becomes greater the nearer it is placed to the sink. Cutting near the sink of a forward dependence places the sink early in a thread and does not allow sufficient time for the source to execute. Similarly, cutting near the sink of a backward dependence places the source late in a thread,

which has the same effect. Our dependence penalty attempts to approximate the synchronization delay necessary for the threads to execute correctly.

Edge weights between the source $P$ and the sink $C$ are increased by the estimated number of synchonization cycles required if that edge is cut. This is illustrated in Figure 2.2. The two size calculations are performed as in Section 2.3.5 with the cut considered as the end of one thread and the beginning of the other. If positive, the delay value is the estimated number of synchronization cycles; if negative, it is considered to be zero. $\mathcal{D}_i$ is the maximum of all delays caused by dependences crossing $e_i$.

The delay value is an approximation, since thread sizes are estimated and since the only information about the relative starting time of the second thread is that it must begin after the first. Note that, while this means that the actual direction of the dependence is unknown, any starting delay of the second thread assists in supplying the synchronization delay–possibly eliminating the need for it entirely. Our penalty therefore models the potential number of lost cycles, which may be more than the actual number. Influencing the cutting algorithm to avoid any potential lost cycles still serves the intended purpose, though it is conservative.

Setting the dependence penalty for cutting an edge involves one size calculation for the first thread, and then an additional size calculation for each dependence crossing the cut. Therefore the operation is $\Theta(DVE)$, where $D$ represents the number of dependences, and simplifies to $\Theta(DV^2)$.

### 2.3.7 P: *Prediction Penalty*

$\mathcal{P}_i$, the thread prediction penalty due to cutting edge $e_i$, is estimated from branch frequencies and thread sizes. The hardware must predict the successor for each thread. If the prediction is incorrect, then the chosen successor will end up being rolled back, and hence its execution represents overhead. Heavily-biased branches, for example, those that are taken 90% or 10% of the time, are predictable; whereas a branch

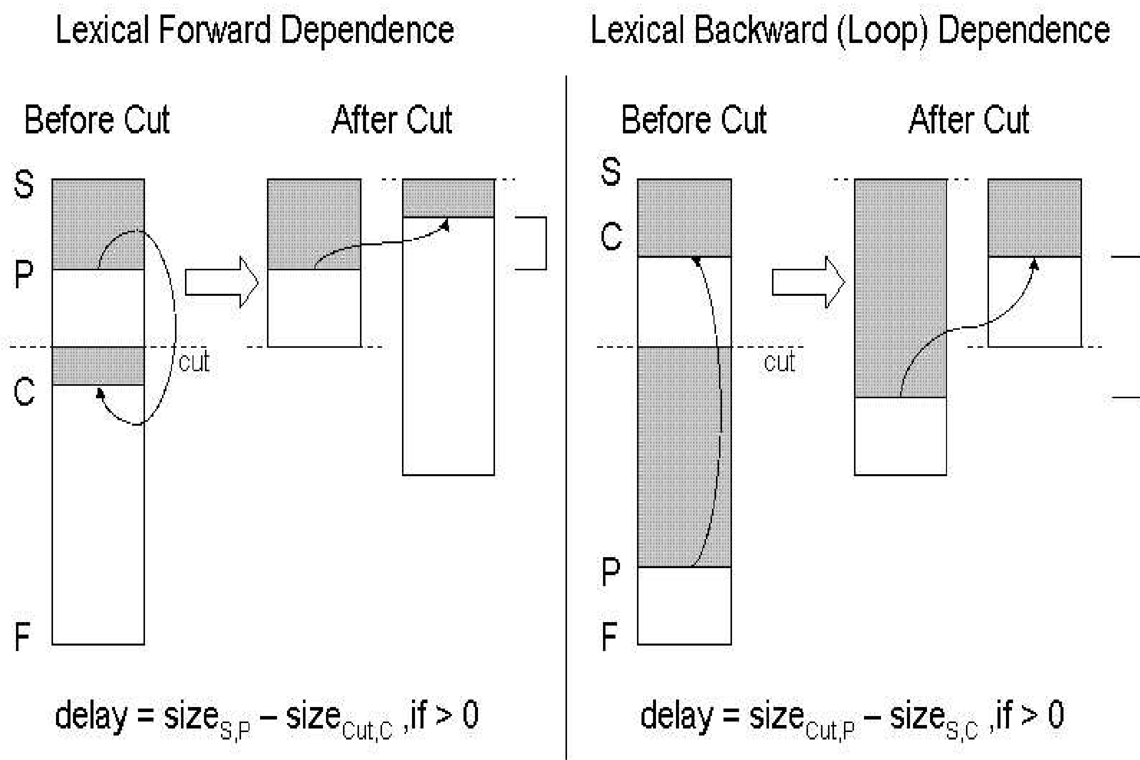


Fig. 2.2. Effect of cutting a dependence edge within a thread

taken 50% of the time is not. When thread boundaries are placed on the edges of unpredictable branches, it is more difficult for the hardware to correctly predict the next thread.

The maximum number of cycles lost due to a thread misprediction is the size of the nonspeculative thread. A speculative thread, by definition, could not have started earlier than the nonspeculative thread, so we have a starting bound. Thread misprediction can only be detected when the nonspeculative thread completes, so we have an ending bound.

The number of cycles actually lost due to thread misprediction at a particular thread boundary can be estimated as a function of the nonspeculative thread size and the extent to which the branch is unpredictable. We use the approximation

that a completely biased branch (0% or 100% taken) is perfectly predictable, while a completely unbiased branch (50% taken) is totally unpredictable. In between we assume that predictability is linear:

$$E_{Cycles\ Lost} = size(nonspec) * unpred(bfreq) \tag{2.3}$$

$$unpred(bfreq) \equiv 1.0 - 2 * |0.5 - bfreq| \tag{2.4}$$

Equation (2.3) allows us to compute an expected value for the number of cycles lost. This value is added to the weight of each edge of the branch. Setting this penalty for all edges requires $E$ steps with a $\Theta(V)$ thread size computation for each, so it is a $\Theta(VE)$ process that simplifies to $\Theta(V^2)$.

### 2.3.8 Profitable: ***When to cut?***

Profitable($C$, $T$) determines if it is better to apply cut $C$ or to keep $T$ intact. The tradeoff is whether the work done by an original thread $T_0$ can be accomplished faster by cutting it into a thread $T_1$ followed by another thread $T_2$. $T_2$ is a weighted composite of successors following $T_1$ that were created by $C$, as previously used in Section 2.3.4 to compute $\mathcal{L}(C)$. Assuming that $T_1$ and $T_2$ execute in parallel and begin at the same time, we should apply the cut $C$ if $T_0$ would require more time than the longer of $T_1$ and $T_2$, plus Cost($C$). However, our approximations do not consider the following. Leaving large threads in a program risks rolling back large amounts of work if there are profiled dependences not present in the experimental run. The efficiency of the hardware implementation and the accuracy of its predictors are not modeled. Therefore, we introduce a function, $\gamma$, that makes large threads appear even larger, effectively requiring a lower tradeoff and preventing the algorithm from being overly optimistic with respect to leaving large threads in the program. By experimentation, we determined that amplifying threads larger than 100 cycles by 20% works well.

$$\gamma(\mathcal{S}(T_0)) \geq \max(\mathcal{S}(T_1), \mathcal{S}(T_2)) + Cost(C) \tag{2.5}$$

## 2.4 Compiler Implementation

Our compiler is based on the GNU C compiler [27] and has been extended several times to add and improve support for speculative CMPs [28, 29]. The f2c program allows the compiler to handle Fortran 77 as well as C code [30]. Compilation involves two passes. The first pass performs common optimizations and unrolls loops with small bodies. The output of the first pass is a CFG for each source file. Our algorithm reads this graph, annotates it with dynamic profiling data, inserts thread boundaries as described throughout this chapter, and outputs a new CFG that reflects its decisions.

The format of the CFG file (also referred to in some of the Multiscalar papers as the partition file) is fairly simple. It is a text file with the first number of each line acting as a code that identifies what information that line conveys. There may be several parameters following the initial code until the next newline. The various codes and the parameters of interest to the algorithm are shown below in Table 2.1. The file format was changed slightly from that used by the original Multiscalar compiler in order to expose more information about the CFG.

The line numbers represent the starting and ending source line numbers for each basic block. The number of instructions is a static count. Block identifiers are unique numbers within a procedure, so that branch destinations can be represented by the identifier of the target block. The type of branch is a number representing either no branch, a conditional branch, or an unconditional branch. Branch information allows the graph edges to be reconstructed from the file. The spec parameter allows the algorithm to indicate whether the edges leaving a block are cut. For function calls, the spec parameter indicates whether the call is included in or excluded from the previous thread. The algorithm only modifies the spec parameters.

The second pass of the compiler reads the new CFG and generates Multiscalar code. Threads are placed in the binary according to the thread boundaries specified on the CFG. The Multiscalar binary format is identical to that of the standard MIPS

Table 2.1
CFG File Format

| Initial Code | Code Name | Parameters |
|---|---|---|
| 0 | FILENAME | name |
| 1 | FUNCNAME | name, insn_count |
| 2 | FUNCCALL | lineno_start, lineno_end, insn_count, block_id, name, spec |
| 9 | LOOPBEGIN | lineno_start, lineno_end, insn_count, block_id, branch_type, branch_dest, spec |
| 10 | LOOPEND | lineno_start, lineno_end, insn_count, block_id, branch_type, branch_dest, spec |
| 11 | FUNCEND | none |
| 13 | BASICBLOCK | lineno_start, lineno_end, insn_count, block_id, branch_type, branch_dest, spec |

binary format with the exception of the thread boundaries. At a high level, the binary is a sequence of threads, where each thread looks like a thread header followed by some executable code. The header of a thread specifies the positions of its potential successor threads within the binary, as well as which register values must be forwarded to the successor thread. A Multiscalar binary is therefore larger than a standard MIPS binary for the same program. The binary can be executed on a cycle-accurate simulator to determine performance and generate the statistics discussed in Section 3.

### 2.5 Obtaining Profile Information

We developed a source code instrumentation tool to facilitate the run-time detection of data dependences. Similar to the method used in [31], executing the instrumented program compares time stamps of all read and write references to produce a

list of dependences at the source-level, including the dependence distance for loop-carried dependences. The distance is important because a dependence crossing more iterations than there are processors is implicitly enforced by the execution model. Anti and output dependences are also irrelevant, as explained in Section 1.3.2.

Source code instrumentation provides branch frequencies, however the profile may not cover all paths taken in the experimental run. For branches lacking this information, we use an estimate that 60% of forward branches are taken and 85% of backward branches are taken [32]. All of the data gathered by instrumentation can be mapped onto the CFG, because we know the line numbers for the basic blocks.

Finally, we obtained the average cycles for each function call by profiling on an SGI Origin system. This system has a MIPS instruction set, similar to the Multiscalar architecture. We used the *pixie* and *prof* profiling tools to record inclusive cycles for each function, such that the average includes any cycles from other calls within that function.

As is typical for profile-guided optimizations, the input data for the profile run differs from the input data used to evaluate our techniques. For all profile runs, we use the `train` data set, and for all experimental runs we used the `ref` data set. This is consistent with the SPEC benchmarking rules (*www.spec.org/osg/cpu2000/docs/runrules.html*). Using the same input for the profile and experimental runs would lead to inflated results [33].

# 3. ANALYSIS OF SPEC CPU2000

## 3.1 Simulator Configuration

All results are obtained using the Multiscalar simulator, configured according to Table 3.1. The memory system parameters are similar to an IBM Power4 [34]. We execute past each program's startup code with a functional simulation before performing a detailed timing simulation for at least 500 million instructions. Rather than using a fixed number of instructions for detailed simulation, we use the same start and end points for all code versions of a program. This is important, since the instruction count of the different versions may vary due to the number and location of thread boundaries. To verify each benchmark passes the validation test, we complete the runs using the functional simulation. A typical simulation window runs for 12 to 24 hours.

## 3.2 Performance

Table 3.2 shows the baseline instructions-per-cycle (IPC) of each benchmark run as a single-thread on one processor and the original speedup on four processors using the techniques in [2]. Each processor is dual-issue out-of-order. Single-thread IPC (instructions per cycle) is typically in the 0.50 to 1.50 range, though mcf is 0.22 and ammp is 0.15 due to extremely poor cache behavior. Four-processor performance is good for applu, art, mgrid, and swim in FP2000, but the rest of the benchmarks remain below a speedup of 1.75. Of particular interest are the INT2000 benchmarks, of which most remain below a speedup of 1.40. Integer (i.e., non-numeric) programs are known to be more difficult for compilers to parallelize than scientific programs, due to the lack of large, regular loops. Our results show similar trends.

Table 3.1
Simulator Configuration

| | |
|---|---|
| CPUs | 4 dual-issue, out-of-order |
| L1 i-cache | 64KB, 2-way, 2-cycle hit |
| L1 d-cache | 64KB, 2-way, 3-cycle hit |
| | 16-byte block, byte-level |
| | disambiguation |
| Rollback Buffer | 64 entries |
| Reorder Buffer | 32 entries |
| Load/Store Queue | 32 entries |
| Function Units | 3 Int, 1 FP, 1 Mem |
| Branch Predictor | path-based, 2 targets |
| Thread Predictor | path-based, 4 targets |
| Descriptor Cache | 16KB, 2-way, 1-cycle hit |
| Shared L2 | 2MB, 8-way, 64-byte block, |
| | 12-cycle hit and transfer |
| L1/L2 Connect | Snoopy split-transaction bus, |
| | 128-bit wide |
| Memory Latency | 80 cycles |

Figures 3.1 and 3.2 show the improvement gained by our decomposition method for the 17 benchmarks. The FP2000 codes that have the highest (mgrid and swim) and lowest (ammp) original performance show insignificant gains from our algorithm. Equake and wupwise show large improvements. The INT2000 codes that show the most improvement are gap and mcf. On average the gain is 13.8% for floating-point programs and 11.3% for integer programs. The gain varies widely, as we compare to the previous, heuristic-based approach, which performs inconsistently. We verified

Table 3.2
Baseline vs Improved Performance

| FP2000 | Single-Thread IPC | Original Speedup | Improved Speedup |
|---|---|---|---|
| ammp | 0.15 | 1.10 | 1.11 |
| applu | 0.86 | 2.32 | 2.49 |
| art | 0.55 | 2.32 | 2.52 |
| equake | 1.07 | 1.30 | 1.54 |
| mesa | 1.43 | 1.26 | 1.36 |
| mgrid | 0.96 | 4.09 | 4.19 |
| sixtrack | 1.04 | 1.22 | 1.30 |
| swim | 0.50 | 4.53 | 4.44 |
| wupwise | 0.91 | 1.74 | 3.49 |
| INT2000 | Single-Thread IPC | Original Speedup | Improved Speedup |
| bzip2 | 1.33 | 1.29 | 1.29 |
| gap | 1.26 | 1.32 | 1.47 |
| gzip | 1.28 | 1.37 | 1.37 |
| mcf | 0.22 | 1.05 | 1.85 |
| parser | 1.07 | 1.09 | 1.16 |
| twolf | 1.00 | 1.22 | 1.24 |
| vortex | 1.36 | 1.59 | 1.66 |
| vpr | 1.21 | 1.47 | 1.56 |

our algorithm performs as well as manual thread selection for some of the smaller benchmarks.

A more detailed analysis showed that most of the improvement in FP2000 was from using the dependence profile to identify potentially parallel loops, whereas most of the improvement in INT2000 was from thread decomposition. For equake and wupwise, a few significant outer loops were parallel, which lead to large improvements.

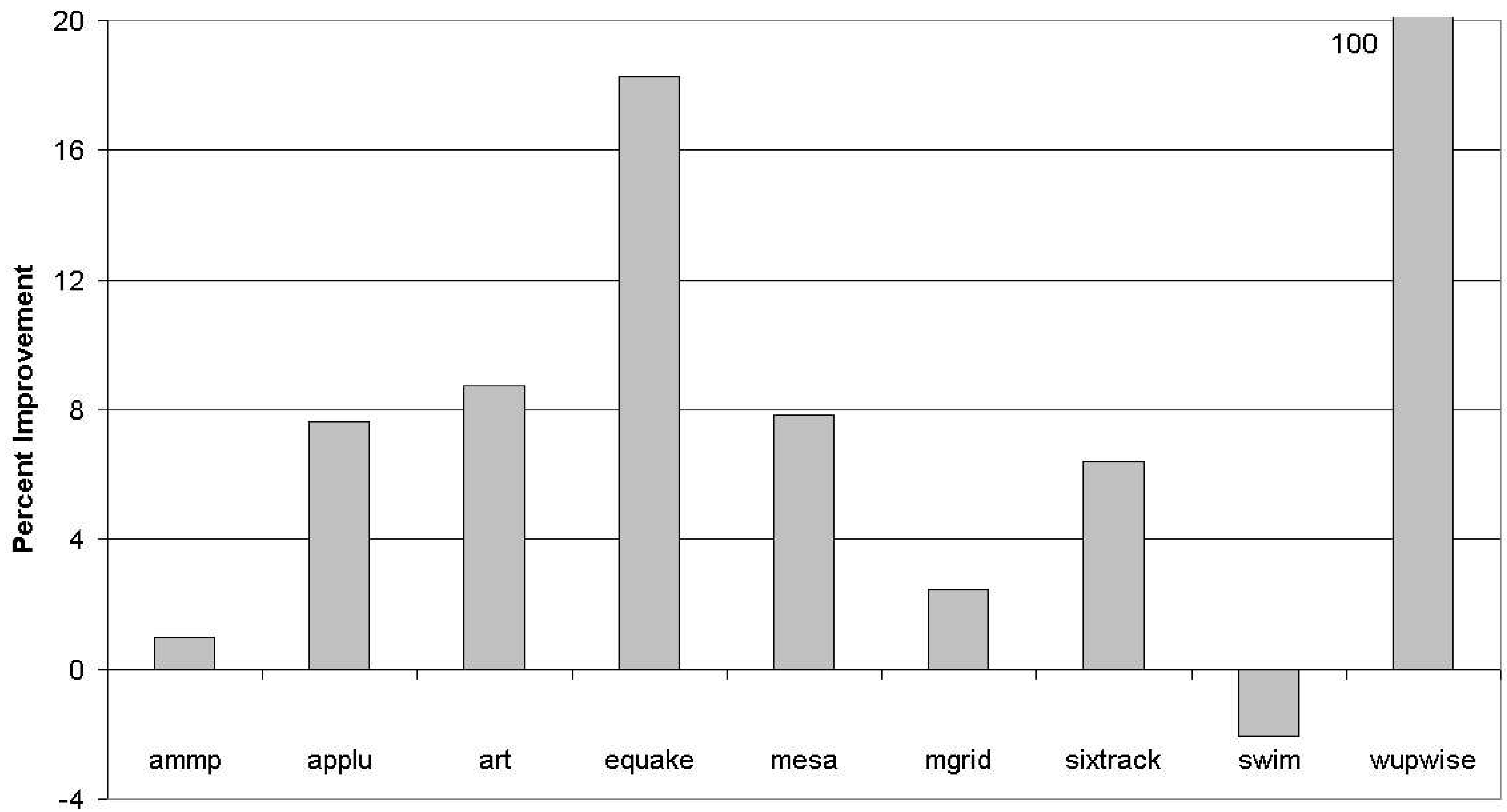


Fig. 3.1. SPEC FP2000 Improvement of our min-cut approach over the best known method

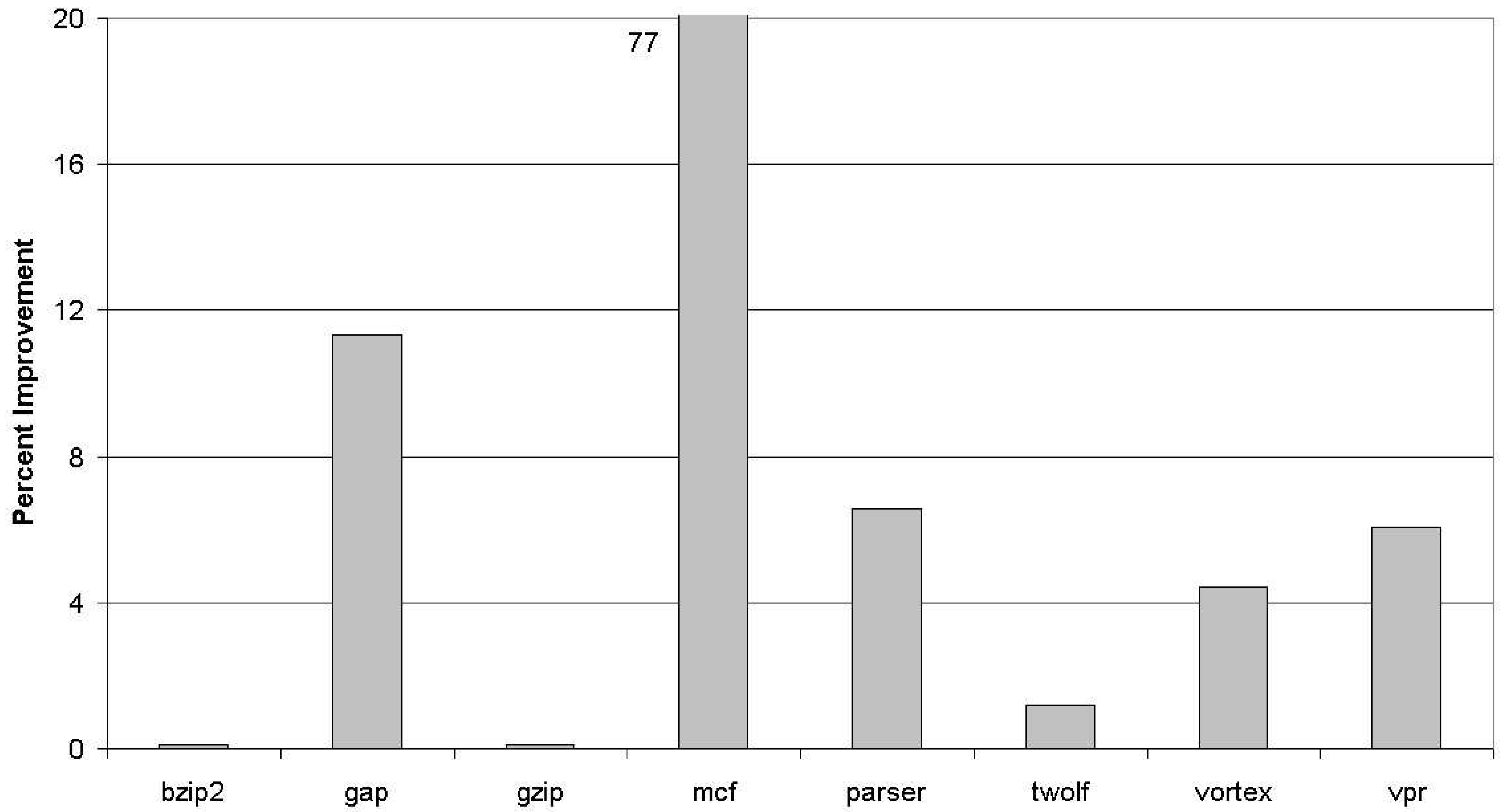


Fig. 3.2. SPEC INT2000 Improvement of our min-cut approach over the best known method

### 3.3 Overhead Analysis

Figures 3.3 and 3.4 show the relative importance of the different overheads for each benchmark, first for the original thread selection and then using our thread decomposition method. The 100% mark represents the total amount of overhead for the original method for each benchmark.

Load imbalance decreases in applu, equake, and wupwise. Misprediction overhead decreases in equake and sixtrack. In mesa, threading overhead and dependence overhead decrease while misprediction overhead increases; however, the overall overhead decreases due to the better tradeoff. Gap mainly improves due to load balancing and mcf shows a significant reduction in dependence overhead. Thread prediction improves in parser, and vpr becomes more balanced.

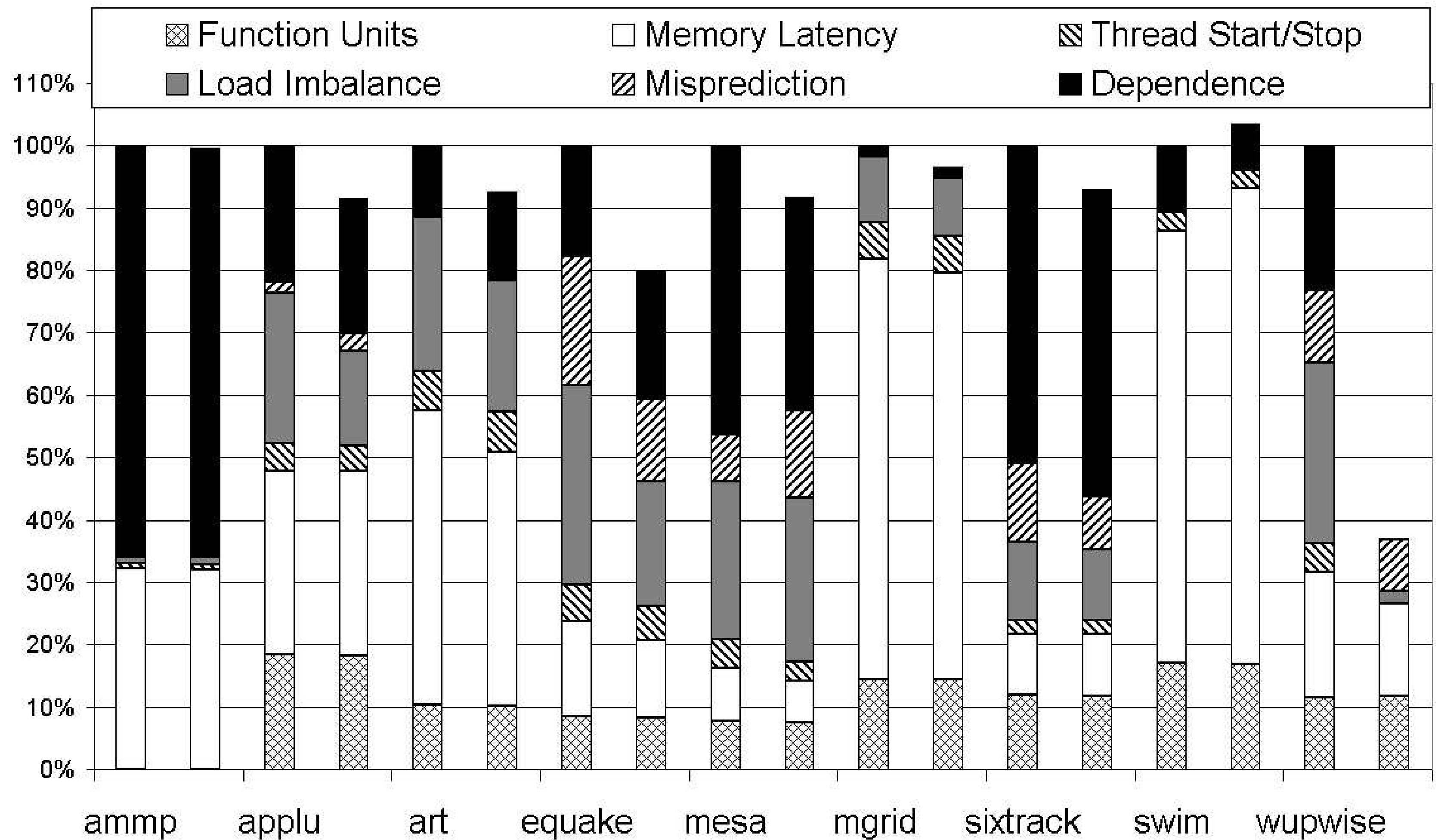


Fig. 3.3. SPEC FP2000 Relative importance of reduced overheads compared to 100% of original overhead

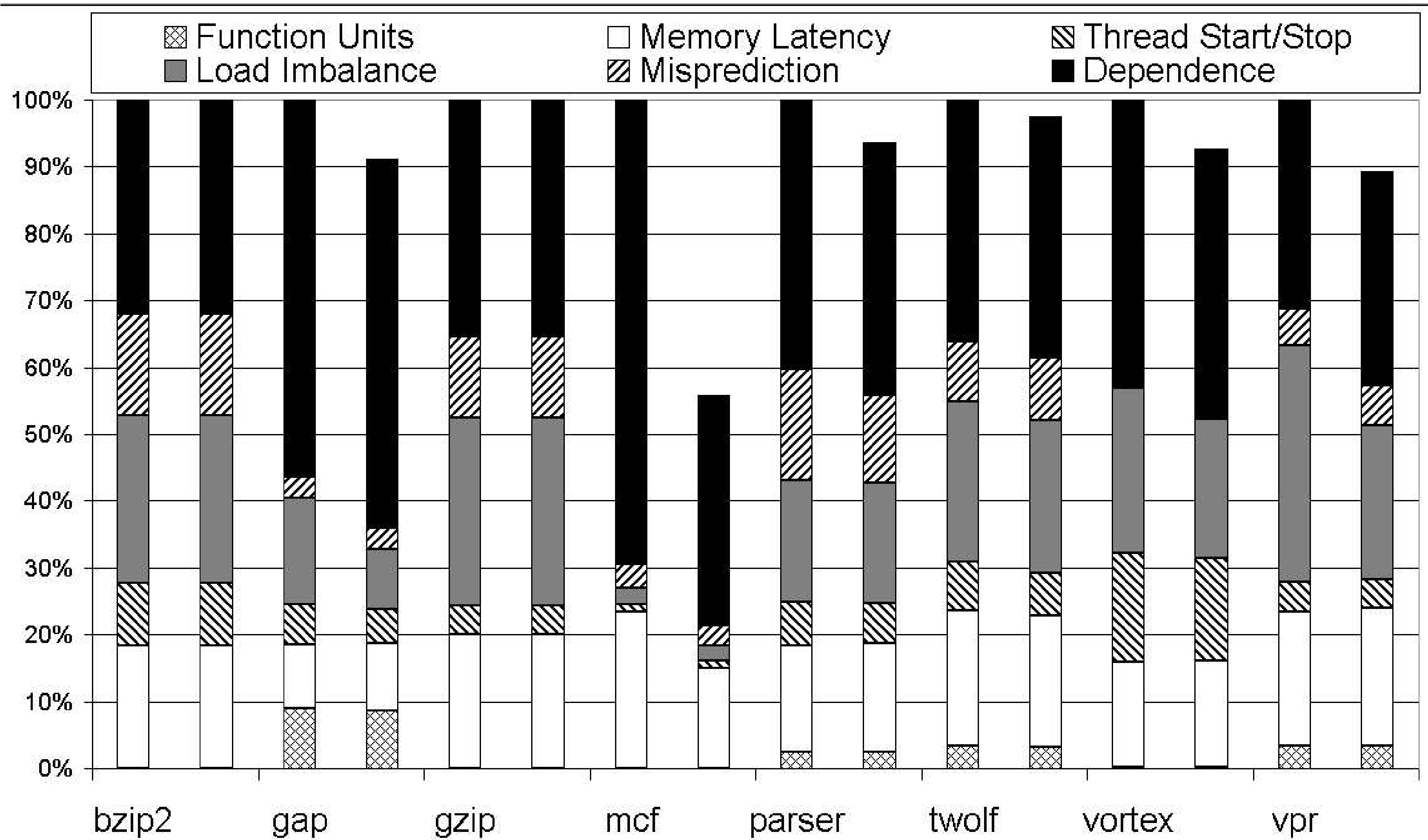


Fig. 3.4. SPEC INT2000 Relative importance of reduced overheads compared to 100% of original overhead

Overheads unrelated to speculation include memory latency and function-unit stalls. They are the dominant overheads for programs that appear fully parallel, such as mgrid and swim. Most noticeably, memory latency increases in swim, leading to a small performance decrease in this code. Memory latency is not modeled in our algorithm and its change due to decomposition, though minor, is influenced by processor affinity – threads accessing the same data being dispatched to the same processor. The current Multiscalar architecture does not allow flexibility in thread-to-processor mapping.

### 3.4 Algorithm Efficiency

We recorded in Table 3.3 how much time was required to compile each benchmark on a 450 MHz Sparc using decomposition versus using the heuristics in [2]. Seven of the benchmarks finished in less than one minute, while 5-10 minutes was common for the rest. Sixtrack contains three extremely large procedures (`daten`, `maincr`, and `umlau6`) that each have over 2000 basic blocks and require a long time for either method. Due to the nature of our algorithm, an extremely large program with typically small functions will decompose faster than a smaller program with several very large functions.

Table 3.3
Basic Blocks Per Procedure (BPP) and Elapsed Time for Heuristics vs Decomposition

| FP2000 | Lines | Avg BPP | Max BPP | Heur(s) | Decomp(s) |
|---|---|---|---|---|---|
| ammp | 13483 | 35 | 492 | 144 | 316 |
| applu | 4975 | 62 | 169 | 186 | 220 |
| art | 1270 | 32 | 137 | 10 | 22 |
| equake | 1513 | 27 | 220 | 22 | 28 |
| mesa | 58724 | 23 | 786 | 394 | 473 |
| mgrid | 1270 | 30 | 72 | 11 | 14 |
| sixtrack | 89918 | 112 | 2996 | 3542 | 5679 |
| swim | 907 | 30 | 73 | 9 | 9 |
| wupwise | 3353 | 36 | 300 | 33 | 29 |
| INT2000 | Lines | Avg BPP | Max BPP | Heur(s) | Decomp(s) |
| bzip2 | 4649 | 24 | 328 | 24 | 58 |
| gap | 71363 | 43 | 1015 | 412 | 907 |
| gzip | 8616 | 28 | 181 | 40 | 76 |
| mcf | 2412 | 27 | 83 | 12 | 15 |
| parser | 11391 | 22 | 343 | 61 | 112 |
| twolf | 20459 | 61 | 451 | 283 | 665 |
| vortex | 67213 | 25 | 614 | 307 | 742 |
| vpr | 17729 | 24 | 337 | 76 | 130 |

# 4. RELATED WORK

Partitioning using standard min-cut or other techniques such as simulated annealing has been applied to compiler problems in the past [35,36]. Balanced min-cut, adapted from [21], has been used to solve VLSI CAD problems, but has not been applied to a compiler problem.

We concentrate on automatically decomposing entire programs into threads, as discussed in [2], which used several heuristics in isolation. In contrast, our algorithm is the first to address all of the overheads simultaneously within a common framework, instead of applying a collection of techniques, which may make conflicting decisions and require careful pass-ordering.

Compilers for CMPs other than Multiscalar have focused exclusively on creating threads from loop bodies or entire function calls, while performing much of the work manually. Some compilers [9] are responsible for adding extra code to support speculation and synchronization, depending on the level of architectural support, whereas we are concerned only with thread boundary placement. Source-code transformations designed specifically to improve speculation, such as those applied manually in [37], can be applied while performing thread selection. We assume that all transformations have been applied prior to decomposition, although we did not modify any of the source code for this paper. If source code is unavailable, it is still possible to make an existing binary run on a speculative CMP by annotating the binary with thread boundaries. The primary disadvantage is that although register dependences are visible, memory dependences are not easily identified. Such approaches have focused on loop parallelism [38]. Thread partitioning can be done entirely in hardware, but suffers from a lack of compile-time information and increased run-time overhead [39].

# 5. CONCLUSIONS

We have presented an algorithm for decomposing a program into threads that tries to achieve maximum parallelism when executed by a speculative chip multiprocesor. The issue of program decomposition for speculative execution arises in all modern speculative processors. We have compared our scheme with the best, previous thread selection method, developed for the Multiscalar architecture. Compared to this heuristic-based approach, our algorithm provides a more rigorous solution. It employs a balanced min-cut algorithm applied to the program's CFG, which we annotate with execution overheads incurred for inserting thread boundaries on its edges.

Our algorithm provides an approximate solution to the NP-hard thread decomposition problem that we believe to be as formal as possible. To reduce complexity, we made a number of approximations. Each decomposition step divides a thread into balanced halves by placing thread boundaries across all edges joining them. In an optimal solution, not every boundary may be desireable. We assign edge weights that represent execution overheads using intuitive approximations. To adjust for some of the inaccuracies introduced by these approximations and to avoid overly-eager speculation, we have made use of a $\gamma$ function to increase bias towards cutting large threads.

Our results show an average improvement of 12.6% over the heuristic-based approach for integer and floating-point programs. While this is a significant gain, an important question is how close our numbers are to optimal decomposition. Since the problem is NP-hard, we manually experimented with the smaller benchmarks (bzip2, equake, mcf, and mgrid) to explore such performance bounds, and we found that our algorithm achieves performance comparable to manual thread selection.

## LIST OF REFERENCES

[1] Seon Wook Kim, Chong Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: A framework for optimizing speculative execution. *PPOPP*, 36(7):2–11, 2001.

[2] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. *31st International Symposium on Microarchitecture*, December 1998.

[3] W. Blume, R. Doallo, R. Eigenman, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[4] 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. *IEEE Computer*, pages 84–89, December 1996.

[5] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd ISCA*, June 1995.

[6] Jean-Yuan Tsai and Pen-Chung Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *International Conference on Parallel Architecture and Compiler Techniques*, pages 35–46, October 1996.

[7] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[8] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.

[9] J. Greggory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *4th IEEE Symposium on HPCA*, pages 2–13, 1998.

[10] Ye Zhang, Lawrence Rauchwerger, and Josep Torrellas. Hardware for speculative run-time parallelization in distributed shared-memory multiprocessors. In *4th IEEE Symposium on HPCA*, pages 162–173, 1998.

[11] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *4th IEEE Symposium on HPCA*, pages 195–205, February 1998.

[12] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th international conference on ASPLOS*, 1998.

[13] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th ISCA*, pages 13–24, June 2000.

[14] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th ISCA*, pages 1–12, June 2000.

[15] María Jesús Garzarán, Milo Prvulovic, José María Llabería, Víctor Viñals, Lawrence Rauchwerger, and Josep Torrellas. Tradeoffs in buffering memory state for thread-level speculation in multiprocessors. In *HPCA*, 2003.

[16] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of IPDPS*, pages 559–564, May 2000.

[17] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, pages 552–571, May 1996.

[18] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 181–190, November 1994.

[19] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th ISCA*, pages 181–193, June 1997.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, San Francisco, 1979.

[21] Hannah H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12), December 1996.

[22] L. James Hwang and Abbas El Gamal. Min-cut replication in partitioned networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1), January 1995.

[23] Chong Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the ICS*, June 2001.

[24] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[25] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[26] Vivek Sarkar. Determining average program execution times and their variance. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, pages 298–312, 1989.

[27] Richard M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation.

[28] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison, January 1998.

[29] Seon Wook Kim. *Compiler Techniques for Speculative Execution.* PhD thesis, Purdue University, August 2001.

[30] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schryer. A Fortran to C converter. Technical report, AT&T Bell Laboratories, March 1995.

[31] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis. In *Proceedings of the ICS*, pages 107–116, July 1993.

[32] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition.* Morgan Kaufmann, New York, 2002.

[33] David W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN Conference on PLDI*, volume 26, pages 59–70, June 1991.

[34] IBM. IBM e-server Power4 system microarchitecture. Technical report, October 2001.

[35] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970.

[36] Rainer Leupers. Instruction scheduling for clustered VLIW DSPs. In *IEEE PACT*, pages 291–300, 2000.

[37] Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the ICS*, pages 21–30, 1999.

[38] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *Proceedings of the ICS*, pages 85–92, 1998.

[39] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the MEM-slicing algorithm. *Journal of Universal Computer Science*, 6(10), 2000.