IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

TITLE OF PAPER/ARTICLE/REPORT, INCLUDING ALL CONTENT IN ANY FORM, FORMAT, OR MEDIA (hereinafter, "the Work"):

Parabilis: Speeding up Single-Threaded Applications by Extracting Fine-Grained Threads for Multi-Core Execution

COMPLETE LIST OF AUTHORS:

Ademola Fawibe, Oghenekarho Okobiah, Oleg Garitselov, Krishna Kavi, Izuchukwu Nwachukwu, Mohana Asha Latha Dubasi, Vinay R. Prahbu

IEEE PUBLICATION TITLE (Journal, Magazine, Conference, Book):

The 10th International Symposium on Parallel and Distributed Computing (ISPDC 2011)

COPYRIGHT TRANSFER

1. The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the above Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

CONSENT AND RELEASE

2. In the event the undersigned makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the undersigned, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.

3. In connection with the permission granted in Section 2, the undersigned hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

4. The undersigned hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the undersigned has obtained any necessary permissions. Where necessary, the undersigned has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE.

☐ Please check this box if you do not wish to have video/audio recordings made of your conference presentation.

See reverse side for Retained Rights/Terms and Conditions, and Author Responsibilities.

GENERAL TERMS

- The undersigned represents that he/she has the power and authority to make and execute this assignment.
- The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
- In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall become null and void and all materials embodying the Work submitted to the IEEE will be destroyed.
- For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.

(1)____________________ May 7th, 2011

Author/Authorized Agent for Joint Authors **Date**

U.S. GOVERNMENT EMPLOYEE CERTIFICATION (WHERE APPLICABLE)

This will certify that all authors of the Work are U.S. government employees and prepared the Work on a subject within the scope of their official duties. As such, the Work is not subject to U.S. copyright protection.

(2)____________________ ____________

Authorized Signature **Date**

(Authors who are U.S. government employees should also sign signature line (1) above to enable the IEEE to claim and protect its copyright in international jurisdictions.)

CROWN COPYRIGHT CERTIFICATION (WHERE APPLICABLE)

This will certify that all authors of the Work are employees of the British or British Commonwealth Government and prepared the Work in connection with their official duties. As such, the Work is subject to Crown Copyright and is not assigned to the IEEE as set forth in the first sentence of the Copyright Transfer Section above. The undersigned acknowledges, however, that the IEEE has the right to publish, distribute and reprint the Work in all forms and media.

(3)____________________ ____________

Authorized Signature **Date**

(Authors who are British or British Commonwealth Government employees should also sign line (1) above to indicate their acceptance of all terms other than the copyright transfer.)

rev. 020711

IEEE COPYRIGHT FORM *(continued)*

RETAINED RIGHTS/TERMS AND CONDITIONS

General

1. Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
2. Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
3. In the case of a Work performed under a U.S. Government contract or grant, the IEEE recognizes that the U.S. Government has royalty-free permission to reproduce all or portions of the Work, and to authorize others to do so, for official U.S. Government purposes only, if the contract/grant so requires.
4. Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use. The IEEE Intellectual Property Rights office must handle all such third-party requests.
5. Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

Author Online Use

6. Personal Servers. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
7. Classroom or Internal Training Use. An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
8. Electronic Preprints. Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to the final, published article in IEEE Xplore.

INFORMATION FOR AUTHORS

Author Responsibilities

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/pub_tools_policies.html. Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

Author/Employer Rights

If you are employed and prepared the Work on a subject within the scope of your employment, the copyright in the Work belongs to your employer as a work-for-hire. In that case, the IEEE assumes that when you sign this Form, you are authorized to do so by your employer and that your employer has consented to the transfer of copyright, to the representation and warranty of publication rights, and to all other terms and conditions of this Form. If such authorization and consent has not been given to you, an authorized representative of your employer should sign this Form as the Author.

IEEE Copyright Ownership

It is the formal policy of the IEEE to own the copyrights to all copyrightable material in its technical publications and to the individual contributions contained therein, in order to protect the interests of the IEEE, its authors and their employers, and, at the same time, to facilitate the appropriate re-use of this material by others. The IEEE distributes its technical publications throughout the world and does so by various means such as hard copy, microfiche, microfilm, and electronic media. It also abstracts and may translate its publications, and articles contained therein, for inclusion in various compendiums, collective works, databases and similar publications.

THIS FORM MUST ACCOMPANY THE SUBMISSION OF THE AUTHOR'S MANUSCRIPT.
Questions about the submission of the form or manuscript must be sent to the publication's editor.
Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966 (telephone)

Parabilis: Speeding up Single-Threaded Applications by Extracting Fine-Grained Threads for Multi-Core Execution

Ademola Fawibe[1], Oghenekarho Okobiah[2], Oleg Garitselov[3], Krishna Kavi[4], Izuchukwu Nwachukwu[5],
Mohana Asha Latha Dubasi[6], and Vinay R. Prabhu[7]
Department of Computer Science and Engineering,
University of North Texas, Denton, TX 76203, USA.
Email: {aof0006[1], oo0032[2], omg0006[3], Krishna.Kavi[4], iun0001[5], md0219[6], vbr0004[7]}@unt.edu

Abstract

The trend in architectural designs has been towards using simple cores for building multicore chips, instead of a single complex out-of-order (OOO) cores, due to the increased complexity and energy requirements of out of order processors. Multicore chips provide better performance when compared with OOO cores while executing parallel applications. However, they are not able to exploit the parallelism inherent in single threaded applications. To this end, this paper presents a compiler optimization methodology coupled with minimal hardware extensions to extract simple fine-grained threads from a single-threaded application, for execution on multiple cores of a chip multiprocessor (CMP). These fine-grained threads are independent and eliminate the need for communication between cores, reducing costly communication latencies. This approach, which we call Parabilis is scalable for up to eight cores, and does not require complex hardware additions to simple multicore systems. Our evaluation shows that Parabilis yields an average speedup of 1.51 on an 8-core CMP architecture.

1. Introduction and Motivation

Powerful out-of-order (OOO) and superscalar processors have dominated the traditional computing space until recently. The complexity of these out-of-order designs significantly increases the transistor count. As the number of transistors on the processor chips increases and moves past beyond the billion mark, some inherent concerns are exposed; including power dissipation, memory capacity limitations and delays due to global wire communication across elements on the processing chip. The complexity of hardware structures needed to extract implicit parallelism in out-of-order (OOO) processors increases power dissipation and the overall complexity of processor design. Therefore, the trend of increasing single processor frequency and complexity of processing elements present in an OOO processor is seen as a not viable option. But there is still a need to provide solutions which address improving the performance of single-threaded applications. Current research efforts are focused on building chip multiprocessors. These multicore chips consist of several simpler cores, with less complex hardware structures. The inherent nature and configurations of CMPs naturally lend themselves to improved performance for multi-threaded and parallel applications. In addition, there has been a large increase in multithreaded and parallel programming to take advantage of the multiple data stream processing power of CMPs. However, a majority of applications available are single-threaded, and single-threaded execution performance is not explicitly addressed by multicore designs. While some compiler techniques exist for extracting parallelism in program loops, the level of parallelization available in some applications is limited. In order to fully extract parallelism from these applications, significant time and effort by programmers would be required. More recently there have been several efforts [1, 2] to improve, adapt or modify CMP hardware for single-threaded performance improvement. Most of these schemes show limited improved performance compared to single core OOO execution, and often require significant extensions to hardware structure in CMPs (thus defeating the purpose of using simple cores).

In this research we take an approach that relies heavily on compilers to extract fine-grained parallelism from single-threaded applications, and executing these threads on several simple cores. This approach requires minimal extensions to the hardware. We call our system

Parabilis. Parabilis extracts fine-grained threads through the use of compiler optimization algorithms that statically schedules independent instructions on different cores in a CMP configuration. However the problem of data dependency limits the number of threads that can be extracted. This can either be overcome by inter-thread communication or instruction replication. We choose to replicate instructions to avoid the overhead of inter-thread communication. We believe that exploiting parallelism at the basic block level results in improved performance for "single-threaded" applications. Parabilis is implemented with minimal hardware modification of a conventional CMP configuration. We add a cross-register communication network, based on idea of the operand network seen in [3], and new state bit to registers to track data dependence and assure program correctness for replicated instructions executed on different cores. We show that Parabilis significantly improves the speed of "single-threaded" applications running on a CMP, and approaches the performance of OOO processors.

The rest of this paper is organized as follows: Section 2 surveys related research in this area. The algorithm and additional hardware modifications needed for Parabilis are discussed in Section 3. Section 4 presents the simulation methodology and experimentation results of our scheme. The results and analysis are presented in Section 5. In Section 6, we highlight factors on the limitations of this work and detail future work needed to overcome these limitations. Finally, Section 7 concludes this paper.

2. Prior Related Research

The prohibitive power consumption and increasing hardware complexity, coupled with the increasing difficulty in improving instruction level parallelism has led to renewed interest in research for alternative solutions. The adoption of CMPs over SMT processors by industry supports this notion. The limitations of CMPs for sequential or single thread based applications have continued to drive the need to adapt CMPs for such applications. The native configuration of CMPs naturally lends itself to improved performances for parallel applications. However, OOO processors still provide better performance than CMPs when executing single-threaded applications. To this end, there has been a significant amount of research on what configuration is best to adopt.

One approach utilized by several proposals employs primarily hardware modifications to explore single-thread performance improvement. In [1], a dual configuration architecture is proposed, in which groups of independent CMPs can be dynamically fused into a more powerful single core. This approach aims to accommodate software diversity by using independent CMPs for parallel applications and fused cores for sequential applications. Similar configurations are also explored in [2, 3]. These approaches tend to ease the constraint of power consumption and simplify the design of complex cores by using multiple simpler in-order cores. While fusing cores does yield performance approaching single OOO processor performance, fusing of cores introduces several additional hardware complexities.

There has also been published research which utilizes the approach of thread decomposition. The single stream of instructions present in a single-threaded applications can be split into smaller threads at varying levels of granularity. In [4], a scheme is proposed where a sequential application is speculatively decomposed into fine-grained threads for execution on CMPs. Since the threads are speculatively decomposed at compile time, threads incur data dependencies, which may be addressed by inter-thread communication or instruction replication. Parabilis is largely inspired by this scheme, however we seek to present an approach to extract finer-grained threads which we claim achieve more thread level parallelism in sequential applications. In addition, [5] presents a work very similar to our paper. Their simulation results are similar to our work, and serve as further justification of the ideas presented. The work in [6] utilizes execution profiling to select candidates from potential mini-threads which have been extracted from single-threaded applications, which may subsequently be compiled into the application itself. In general, this approach introduces the challenges of thread scheduling in addition to load balancing across the cores. Managing data dependencies across threads is another key issue, which is further complicated by the cost of inter-core communication to handle data dependencies across the threads. The research in [7] presents a scheme to address data dependencies across threads by utilizing compiler and hardware support to deduce and insert synchronization instructions into threads created from a single-threaded application.

Loop iterations present another prospective target for thread partitioning, however optimal thread extraction still presents some challenges for loops with loop-carried data dependencies. In addition, pointer references may prevent a compiler from statically extracting loop-level parallelism inherent in an application. The work in [8] is a related scheme in which loops are partitioned into fine-grained threads for execution on multiple cores, while using speculation to handle dependences. Loops which are resistant to compiler optimization due dynamic references can be decomposed into component threads, akin to a producer-consumer model. Using appropriate synchronization, these component threads can be set to execute on different cores. In [9], an algorithm that overcomes the problems of load balancing and inter-core communication is presented. For each loop iteration, [10] speculative threads are spawned, which are squashed if they are dependent on previous iterations. The research in [11, 12] present a Java-based run-time system consisting of a hardware profiler to dynamically profile loops to determine optimal iteration candidates for parallelization. In [11], a dynamic compiler is also used to re-compile selected speculative thread loops for parallel execution.

Instruction replication can be used to further exploit parallelism, which may be employed in concert with thread decomposition to reduce or eliminate inter-core communication and data dependencies. In [13], the authors present an approach utilizing modulo scheduling and instruction replication for loops on clustered micro-architectures while [14] presents an approach that optimizes instruction replication to minimize the overheard of inter-core communication. This approach speculatively decomposes single-threaded applications and employs a balanced min-cut approach that minimize load imbalances while keeping communication overheard low. In [15], a technique for duplicating instructions for execution in VLIW architectures is presented. The work in [9] presents a scheme to selectively replicate instructions in multi-clustered architectures. The clusters are connected through an interconnect to support inter-cluster communication. Both of these techniques primarily utilize compiler support, but additional hardware is needed to enforce instruction dependencies. Speculative CMPs [16, 17], provide hardware structures for this purpose, in addition to temporary storage for, and the ability to roll back speculative execution.

Our work explores finer-grained threads than [4] and we use recursive thread decomposition. In addition, we attempt to minimize load imbalances by examining latency costs while selecting candidate threads, at the same time eliminating inter-thread dependencies by replicating instructions.

3. Parabilis:Algorithm and Hardware Support

Our proposed scheme is called Parabilis, which utilizes compiler support in combination with minimal hardware modifications. The following subsections detail the methodology and hardware modifications necessary for implementing Parabilis.

3.1. Compiler Algorithm Optimizations

The static compile time algorithm extracts fine grain threads through the use of graph traversals. The instructions and data dependencies in a basic block can be represented as a directed acyclic graph (DAG) [18] (an example is shown in figure 1).

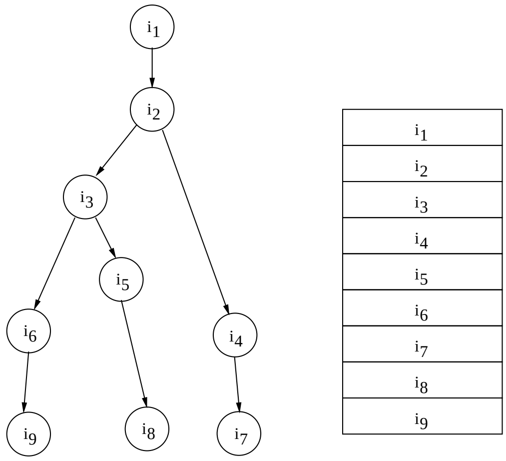


Figure 1. Example of Basic Block and Schedule

Paths from the root node(s) to the leaf nodes represent potential fine-grained threads for parallel execution. The number of leaf nodes dictates the maximum number of parallel threads that can be extracted from a basic block. The figure 2 shows a possible thread extraction of the basic block in figure 1.

The set of vertices and edges representing a path is an incomplete representation of a fine-grained thread. The goal is to obtain a set of edges and vertices such that no edges cross the sub-graph boundary. Therefore, the vertices representing additional instructions are incorporated into the set (i.e., replicated) until this goal is accomplished, which forms a complete thread subgraph. These subgraphs, represented as fine-grained threads, can be scheduled to execute concurrently on different cores. Instructions may appear in multiple subgraphs. The replication of instruction in multiple threads eliminates data dependencies among the threads.

Using replicated instructions, the need to exchange data in registers (corresponding to dependent variables) is eliminated - the each thread will compute its own register values for the dependent variables. However, the replication of instructions may lead to load-imbalance (some threads may have more more instructions to execute), and the need to merge the computed values in registers. We analyze the load imbalances and merging of registers in scheduling fine-grained threads on available cores. Our thread generation algorithm is detailed below 1.

Algorithm 1 Algorithm that implements Parabilis

```
1:  Parse the assembly code.
2:  Extract the basic block.
3:  Find dependencies within instructions within each
    basic block
4:  Each thread starts from each leaf nodes including all
    dependent instructions
5:  Keep the longest thread and combine the short
    threads that give the most repetition to fit under the
    longest limit
6:  for do
7:      If too many threads force merge with smallest
        cost
8:      Longest_Thread + Comm_Overhead =
        BB_Latency
9:  end for
10: while Basic Block not finished do
11:     Move to the next basic block.
12: end while
```

The algorithm starts by identifying and then extracting basic blocks from the code generated by a compiler. After the basic blocks have been extracted, dependent instructions are found. All basic fine-grained threads are extracted by exploring all dependent paths from the leaf nodes to the root. The initial number of fine-grained threads extracted is only limited by the available leaf nodes and not based on the number of cores, creating more fine-grained threads than the number of cores available. We then combine the fine-grained threads in such way that the execution time (or execution latency) of the larger thread does not adversely increase the parallel execution of the fine-grained threads. Note that the execution time of the basic block is limited by the fine-grained thread with maximum number of instructions. We also considered using the number of replicated instruction as criteria for merging threads.

If the number of threads remaining is still more than the available cores, we employ a cost minimizing technique to determine which threads to combine. We exhaustively search for pair-wise thread combinations that yield the best overall execution times (or execution latencies). We repeat this process of pair-wise merging of threads until the number of threads is equal to the number of cores. Figure 3 shows an example using the threads generated in figure 2 with 2 cores. An exhaustive search for merging T_1, T_2, T_3 is done to find which thread combinations yield the best latencies. There are three possible combinations for these three threads, [(T_{12}, T_3), (T_{13}, T_2), (T_{23}, T_1)]. The best combination is(T_{13}, T_2). This example shows that although T_1 and T_2 have more common instructions, but their merger does not yield the best savings. This cost defined in lines 7 and 8 of Algorithm 1 favors greater latency savings, which is our main goal.

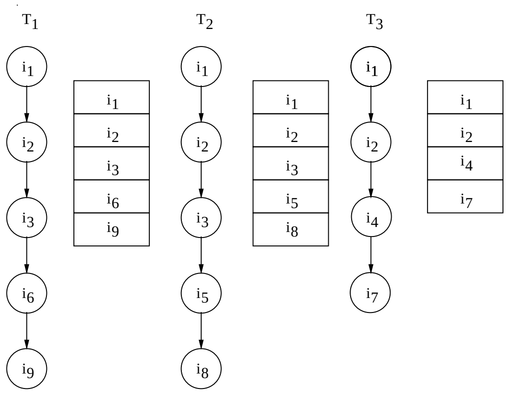


Figure 2. Example of threads extracted from a Basic Block

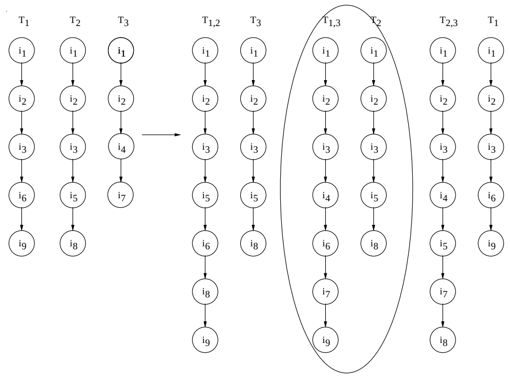
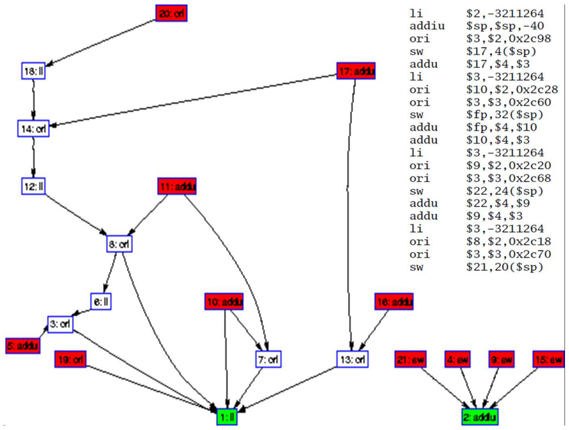


Figure 3. Example of Force Merging

Optimal scheduling is an N-P complete problem and our scheduling algorithm does not always provide the optimal solution. The figure 4 displays an example of a basic block from an actual code in MIPS assembly language and the resulting threads.

The number of thread which can execute concurrently is constrained by the number of available cores. The overall execution time of the basic block would be bound by the execution time of the longest fine-grained thread. At the end of a basic block execution, register values need to be communicated across the cores. To this end, we propose some basic hardware modifications to facilitate this process. These modifications are detailed in the next subsection.



```
li      $2,-3211264
addiu   $sp,$sp,-40
ori     $3,$2,0x2c98
sw      $17,4($sp)
addu    $17,$4,$3
li      $3,-3211264
ori     $10,$2,0x2c28
ori     $3,$3,0x2c60
sw      $fp,32($sp)
addu    $fp,$4,$10
addu    $10,$4,$3
li      $3,-3211264
ori     $9,$2,0x2c20
ori     $3,$3,0x2c68
sw      $22,24($sp)
addu    $22,$4,$9
addu    $9,$4,$3
li      $3,-3211264
ori     $8,$2,0x2c18
ori     $3,$3,0x2c70
sw      $21,20($sp)
```

Figure 4. Basic Block:Thread Extraction and Scheduling

3.2. Hardware

Parabilis extends the basic architecture found in CMPs with additional register metadata and a cross-core register communication network bus. The special cross-core register communication bus is based on the technique used in [3]. A diagram of a multicore Parabilis architecture is shown in figure 5. The figure shows how conventional CMPs cores can be connected with the cross-core register communication network bus. Each core has private L1 (instruction and data) and L2 caches. An expanded view of a single core is also shown in figure 5. It shows how the cross-core register data is communicated to each core through the help of a communication functional unit (Comm FU).

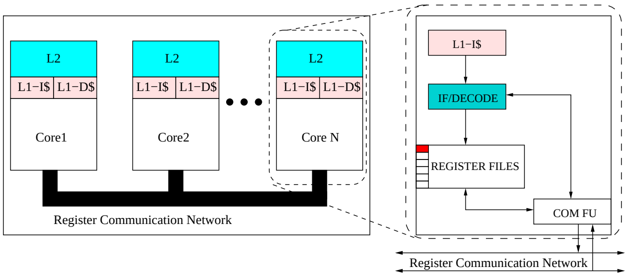


Figure 5. Conceptual Parabilis Architecture

An extra register bit is added to the register metadata to indicate if that register has been modified by that core during the basic block execution. A '1' signifies that the register has been modified and the updated value needs to be communicated to other cores (for use in computations in future basic blocks) at the end of the basic block execution. A '0' signals that the register has not been modified and the current values are valid. After the execution of the basic blocks, all registers with a modified value are broadcast to other cores on the register communication bus. The fast cross-core communication bus is able to communicate a register value in 3 cycles. Our estimates differ from the one in [3] by a cycle. The estimates for communication in [3] add an extra cycle per hop between cores. Our configuration allows for direct communication between all cores. Further details are discussed in the performance evaluation section.

4. Performance Evaluation and Simulation Setup

In evaluating Parabilis, we compare the speedup achieved by Parabilis to that of conventional CMPs and OOO for selected SPEC2006 Benchmarks. We evaluate speed up by comparing the reduction in total cycles needed for execution. In the following subsections, the configuration and simulation setups are discussed.

4.1. Configuration

For Parabilis, we assume simple in-order cores and each core has one functional unit of each type; floating point (FP), Integer (Int), Multiplier (Mult). For the OOO simulation, we assume 2-wide instruction issue architecture with one functional unit of each type: floating point (FP), Integer (Int), Multiplier (Mult). We run simulation tests for up to 64 cores of a CMP.

4.2. Simulation

Our thread generation algorithm is applied to thee MIPS code generated by the GCC compiler for the selected SPEC-2006 benchmarks. We use estimated latencies for each MIPS instruction to compute the execution cycle-times for the benchmark programs. The instruction latencies used in our study are taken from [19]. For register communication after completing a basic block, we assume 3 cycle delay: one cycle to push updated values to the register network, one cycle to propagate values across the network, and one cycle to update the registers. In addition, we model a perfect network with a 0 cycle communication latency to establish an ideal maximum bound for performance.

5. Results and Analysis

Table 1 and Fig. 6 show the results offer simulations, normalized to the execution time of a single in-order

CMP core. We achieved the best performance gains for mcf, even beating the performance of OOO core, when 16 simple cores are used. In addition, milc approaches the performance of an OOO core when using 64 cores The general trend across all benchmarks achieve only small gains beyond the 8 cores. The average speedups are 1.30, 1.44, 1.50, and 1.51 for 2, 4, 8 and 16 cores, respectively.

Table 1. Simulation Results With Communication Overhead

Config	2	4	8	16	32	64	OOO
bzip2	1.119	1.140	1.145	1.145	1.147	1.147	1.682
gcc	1.267	1.359	1.387	1.393	1.398	1.400	1.512
hmmer	1.357	1.556	1.632	1.648	1.661	1.667	1.995
lbm	1.494	1.930	2.160	2.185	2.232	2.244	3.272
mcf	1.320	1.448	1.491	1.498	1.505	1.507	1.487
milc	1.412	1.677	1.793	1.833	1.849	1.855	1.859
sjeng	1.272	1.378	1.408	1.47	1.425	1.427	1.738
specrand	1.099	1.176	1.195	1.195	1.210	1.210	1.557
sphinx3	1.367	1.527	1.579	1.591	1.599	1.602	1.673
Geometric mean	1.293	1.444	1.502	1.513	1.525	1.528	1.809

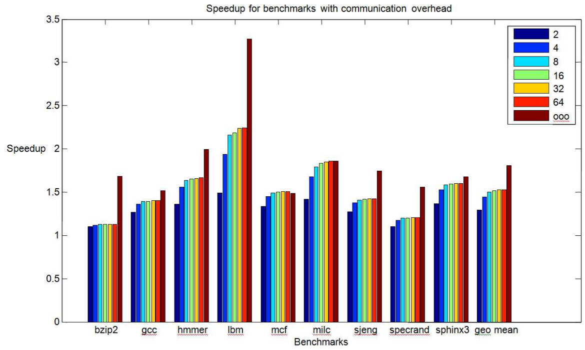


Figure 6. Speedup for Benchmarks with Communication

Table 2 and Fig. 7 shows our results when the communication overhead is set to zero. The results are similar to those obtained with a 3 cycle communication overhead. The average speedup is slightly better at 1.33, 1.49, 1.55 and 1.56 for the 2, 4, 8 and 16 cores respectively. This indicated that even with a perfect bus mechanism the scheme does not exceed the performance of an OOO core.

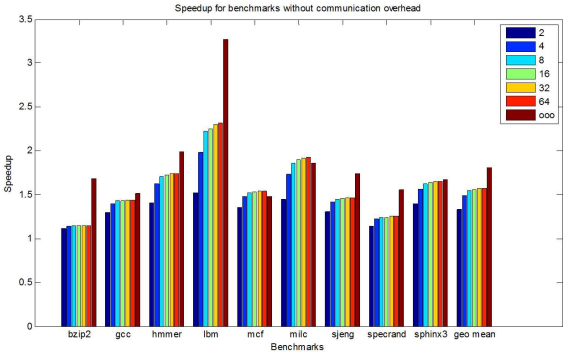


Figure 7. Speedup for Benchmarks without Communication

5.1. Analysis

From the simulation results, we see an improved speedup for applications using simple in-order cores. However, on average the performance is still less than that of a OOO core. For the milc and mcf benchmarks we equal or outperform OOO cores. Even though we do not outperform OOO, we obtain an average speedup of 1.5 using 8 in-order cores, when compared to using a single in-order core. We can further improve the performance achieved by our system using simple compiler techniques like loop unrolling, software pipelining, instruction reordering and variable (register) renaming. We believe that the primary reason for the OOO core performance is their use of register naming (with a large number of renaming registers). This renaming eliminates many data dependencies and permits concurrent execution of instructions (in out of order). We hope to explore register renaming for CMPs in our future work.

6. Future Work and Limitations

6.1. Limitations

Our implemented model as it stands, contains some limitations. The execution stream is a simple traversal of the assembly code produced, and does not account for the factors introduced by control flow changes, particularly the multiple jumps associated with loops. In addition, this scheme does not account for latencies introduced by load or store misses. The variable latencies imposed by network traffic and the memory hierarchy will have a significant impact on the performance gains. Another limitation is the nature of the static scheduling, which forces a the application to conform to a set CMP configuration.

6.2. Future Work

In the future we will explore extensions to our architecture by exploiting register renaming. Register renaming will reduce the number of anti and output dependencies, allowing for the extraction of more finer-grained threads. In particular, the length of the longest thread, based on the number of instructions in the thread, can be shorted since some data dependencies are eliminated with register renaming, and eliminate the need for replicated instructions. In addition, using liveness analysis, the number of registers that need to be updated by the fine-grained threads across basic blocks can be reduced (only the live registers need to be updated). As previously mentioned, accounting for the variable latencies of memory operations is another important factor to investigate.

Another potential factor is the use of selective decomposition of a basic block for execution on multiple cores. If the achieved performance is small, we may execute the code of a basic block on a single core. One other area we can examine is the potential of exchanging register values during the execution of threads of a basic block (and not wait until the end of the basic block). This allows cores to proceed to execute threads from different basic blocks, without waiting for all computations of a basic block to complete. The addition of a priority bit to each register will allow the communication of prioritized values during basic block execution, reducing the need for replication instructions. We also wish to address the limitations imposed by the use of compile time thread generation. An application may be compiled for individual CMP configurations, but hardware resources available to the application may be variable, and adapting the scheduling to account for this variability can improve the scalability and performance. [20] presents a potential basis for work in this area.

Table 2. Simulation Results Without Communication Overhead

Config	2	4	8	16	32	64	OOO
bzip2	1.118	1.140	1.145	1.145	1.146	1.146	1.683
gcc	1.303	1.401	1.430	1.437	1.441	1.444	1.512
hmmer	1.407	1.623	1.705	1.723	1.737	1.744	1.995
lbm	1.525	1.982	2.226	2.253	2.303	2.315	3.272
mcf	1.359	1.482	1.527	1.534	1.542	1.544	1.487
milc	1.453	1.735	1.860	1.903	1.920	1.927	1.859
sjeng	1.306	1.417	1.449	1.458	1.467	1.469	1.738
specrand	1.139	1.220	1.241	1.241	1.258	1.258	1.557
sphinx3	1.401	1.570	1.626	1.638	1.646	1.649	1.673
Geometric Mean	1.328	1.488	1.550	1.562	1.574	1.577	1.809

7. Conclusion

We have presented Parabilis as an alternative approach to the task of increasing the performance of CMPs for executing single-threaded applications. Parabilis addresses this by utilizing compile time analysis to extract fine-grained threads from a single thread and selectively replicating instructions, so that threads can be executed on multiple cores. Parabilis accomplishes this with minimal hardware extensions. The results of our simulations show a 50% speedup over a base simple in-order CMP core. We use this measure since, without OOO, single threaded applications can only run on one simple core. Parabilis achieved up to 83% of the performance of 2-issue Out of Order core. Our study can be improved by accounting for cache misses, and the performance of Parabilis can be improved using (register) renaming, selective replication of instructions, and other compiler techniques such as unrolling loops, software pipelining, instruction reordering.

Acknowledgements

This work is supported in part by the US National Science Foundation Net-Centric Industry/University Cooperative Research Center.

References

[1] E. Ipek, M. Kirman, N. Kirman, and J. F. Martnez, "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," in *ISCA'07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 186–197.

[2] M. B. David Tarjan and K. Skadron, "Federation: Out-of-Order Execution Using Simple In-Order Cores," University of Virginia, Department of Computer Science., Tech. Report CS-2007-11, Tech. Rep., Aug 2007.

[3] H. Zhong, S. A. Lieberman, and S. A. Mahlke, "Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 25–36. [Online]. Available: http://portal.acm.org/citation.cfm?id=1317533.1318098

[4] C. Madriles, P. López, J. M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez, "Boosting Single-thread Performance in

Multi-core Systems through Fine-Grain Multi-Threading," in *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2009, pp. 474–483.

[5] A. Aleta, J. Codina, A. Gonzalez, and D. Kaeli, "Instruction Replication for Clustered Microarchitectures," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec. 2003, pp. 326 – 335.

[6] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Speculative Thread Decomposition Through Empirical Optimization," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 205–214. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229474

[7] X. Wang, Y. Zhao, Y. Wei, S. Song, and B. Han, "Prophet Synchronization Thread Model and Compiler Support," in *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*, Sept. 2010, pp. 81 –87.

[8] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August, "Decoupled Software Pipelining With the Synchronization Array," in *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, September 2004, pp. 177 – 188.

[9] A. Aggarwal and M. Franklin, "Instruction Replication for Reducing Delays Due to Inter-PE Communication Latency," *Computers, IEEE Transactions on*, vol. 54, no. 12, pp. 1496 – 1507, Dec. 2005.

[10] M. Islam, A. Busck, M. Engbom, S. Lee, M. Dubois, and Stenstrom, "Loop-level Speculative Parallelism in Embedded Applications," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, Sept. 2007, p. 3.

[11] M. Chen and K. Olukotun, "The JRPM System for Dynamically Parallelizing Java Programs," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, June 2003, pp. 434 – 445.

[12] ——, "TEST: A Tracer for Extracting Speculative Threads," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, March 2003, pp. 301–312.

[13] A. Aleta, J. Codina, J. Sanchez, A. Gonzalez, and D. Kaeli, "AGAMOS: A Graph-Based Approach to Modulo Scheduling for Clustered Microarchitectures," *Computers, IEEE Transactions on*, vol. 58, no. 6, pp. 770 –783, June 2009.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar, "Min-Cut Program Decomposition for Thread-Level Speculation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: ACM, 2004, pp. 59–70. [Online]. Available: http://doi.acm.org/10.1145/996841.996851

[15] J. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. Irwin, "Compiler-Directed Instruction Duplication for Soft Error Detection," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 1056 – 1057 Vol. 2.

[16] J. Steffan, C. Colohan, A. Zhai, and T. Mowry, "A Scalable Approach To Thread-Level Speculation," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, 2000, pp. 1 – 12.

[17] Y. Zhang, L. Rauchwerger, and J. Torrellas, "Hardware for Speculative Run-Time Parallelization In Distributed Shared-Memory Multiprocessors," in *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, Feb 1998, pp. 162 –173.

[18] M. Heffernan and K. Wilken, "Data-Dependency Graph Transformations for Instruction Scheduling," *Journal of Scheduling*, vol. 8, pp. 427–451, October 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1077342.1077363

[19] J. Heinrich, *MIPS R10000 Microprocessor User's Manual*. [Online]. Available: http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf

[20] Y. Ding, M. Kandemir, P. Raghavan, and M. Irwin, "A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1 –14.