©2010 International Journal of Computer Applications (0975 - 8887)
Volume 1 – No. 16

A Modern Parallel Register Sharing Architecture for Code Compilation

Rajendra Kumar,
Vidya College of Engineering, Meerut (UP)

Dr. P. K. Singh,
MMM Engineering College, Gorakhpur

ABSTRACT

The design of many-core-on-a-chip has allowed renewed an intense interest in parallel computing. On implementation part, it has been seen that most of applications are not able to use enough parallelism in parallel register sharing architecture. The exploitation of potential performance of superscalar processors has shown that processor is fed with sufficient instruction bandwidth. The fetcher and the Instruction Stream Buffer (ISB) are the key elements to achieve this target. Beyond the basic blocks, the instruction stream is not supported by currents ISBs. The split line instruction problem depreciates this situation for x86 processors. With the implementation of Line Weighted Branch Target Buffer (LWBTB), the advance branch information and reassembling of cache lines can be predicted by the ISB. The ISB can fetch some more valid instructions in a cycle through reassembling of original line containing instructions for next basic block. If the cache line size is more than 64 bytes, then there exist good chances to have two basic blocks in the recognized instruction line.

The code generation for parallel register share architecture involves some issues that are not present in sequential code compilation and is inherently complex. To resolve such issues, a consistency contract between the code and the machine can be defined and a compiler is required to preserve the contract during the transformation of code. In this paper, we present a correctness framework to ensure the protection of the contract and then we use code optimization for verification under parallel code.

Keywords

ILP, multithreading, fine-grained, Inthreads, ISB

1. Introduction

Instruction-level parallel processing (ILP) has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write the application. Instruction-level parallelism allows a sequence of instructions derived from a sequential program to be parallelized for execution on the processors having multiple functional units. There are several reasons why the parallelization of a sequential program is important.

The most frequently mentioned reason is that there are many sequential programs that would be convenient to explore the architectural parallelism available with the processor. There are, however, two other reasons that are perhaps more important. First, powerful parallelizers should facilitate programming by allowing the development of much of the code in a familiar sequential programming language such as C. Such programs would also be portable across different classes of machines if effective compilers were developed for each class. The second reason is that it exploits parallelism without requiring the programmer to rewrite existing applications. ILP's success is due to its ability to overlap the execution of individual operations without explicit synchronization.

Consequently, microprocessor increasingly support coarser thread based parallelism in the form of simultaneous multithreading (SMT) [6] and chip multiprocessing (CMP) [12].

Medium to low parallelism is targeted by Inthreads architecture between the ILP and its multithreaded execution. At this end thread based parallelism is applied at a fine granularity by providing extremely light weight threads. A programming model is defined by Inthreads, which share the context of threads to the maximal possible extent including most of the architectural registers and memory address space.

The implicit assumption based on compiler employ many of the code transformations. The registers are local for each thread, and the correctness is broken if they are applied to multithreaded code sharing registers [2]. The compilation techniques necessary to preserve the correctness are also described in this paper.

2. Related work

By applying Amdahl's formulation to the programs in the PARSEC and SPLASH-2 benchmark suites, the applications may not have enough parallelism for modern parallel machines. However, value prediction techniques may allow the parallelization of the sequential portion by predicting values before they are produced. [16] extends Amdahl's formulation to model the data redundancy inherent to each benchmark. The analysis in [16] shows that the performance of PARSEC suite benchmarks may improve by a factor of 180.6% and 232.6% for the SPLASH-2 suite, compared to when only the intrinsic parallelism is considered. This demonstrates the immense potential of fine-grained value prediction in enhancing the performance of modern parallel machines.

Many commercially available embedded processors are capable of extending their base instruction set for a specific domain of applications. While steady progress has been made in the tools and methodologies of automatic instruction set extension for configurable processors, recent study has shown that the limited data bandwidth available in the core processor (e.g., the number of simultaneous accesses to the register file) becomes a serious performance bottleneck. [10] proposes a new low-cost architectural extension and associated compilation techniques to address the data bandwidth problem. [10] also present a novel simultaneous global shadow register binding with a hash function generation algorithm to take full advantage of the extension. The application of this approach leads to a nearly-optimal performance speedup (within 2% of the ideal speedup).

To balance multiple scheduling performance requirements on parallel computer systems, traditional job schedulers are configured with many parameters for defining job or queue priorities. Using many parameters seems flexible, but in reality, tuning their values is highly challenging. To simplify resource management, [19] proposes goal-oriented policies,

which allow system administrators to specify high-level performance goals rather than tuning low-level scheduling parameters.

EPIC (Explicitly Parallel Instruction Computing) architectures, exemplified by the Intel Itanium, support a number of advanced architectural features, such as explicit instruction-level parallelism, instruction predication, and speculative loads from memory. [14] describes techniques to undo some of the effects of such optimizations and thereby improve the quality of reverse engineering such executables.

In [7] a compilation framework is presented that allows the compiler to maximize the benefits of predication as a compiler representation while delaying the final balancing of control flow and predication to schedule time. In [18] a heuristic is presented developed by using the Trimaran simulator. The results are compared to those of the current hyperblock formation heuristic. Weaknesses of the heuristic are exploited and further development is examined. [13] explores how to utilize loop cache to relieve the unnecessary pressure placed on the trace cache by loops. [8] introduces a technique to enhance the ability of dynamic ILP processors to exploit (speculatively executed) parallelism.

[17] presents a performance metric that can be used to guide the optimization of nested loops considering the combined effects of ILP, data reuse and latency hiding techniques. [20] represents the impact of ILP processors on the performance of shared-memory multiprocessors, both without and with the latency hiding optimization of software pre-fetching.

One of the critical goals in code optimization for Multiprocessor-System-on-a-Chip (MPSoC) architectures is to minimize the number of off-chip memory accesses. [9] proposes a strategy that reduces the number of off-chip references due to shared data. It achieves this goal by restructuring a parallelized application code in such a fashion that a given data block is accessed by parallel processors within the same time frame so that its reuse is maximized while it is in the on-chip memory space.

3. Inthreads Architecture

The key elements of Inthreads architecture are a fixed number of lightweight thread contexts. These lightweights are executed over a shared register file. The thread scheduling and management to be performed in the hardware internally is supported by the fixed nature of Inthreads model. The threads initiate scheduling for using the registers in such a manner to avoid conflicts, rather they dedicate a subset of registers for local use of each thread. The registers are communicated by threads in the same manner as shared memory locations are communicated by the conventional threads. For safe communication, the architecture incorporates dedicated synchronization instructions which operate on a set of conditional register of 1-bit. Synchronization registers behave as binary semaphores. The summary of these instructions is given in [2].

The programs required by Inthreads model are supposed to be free from data-race in both memory and register access. The model assigns release semantics to thread *inth.start*, *inth.halt*, *inth.set*, and *inth.resume* instructions and the required semantics are *inth.wait* and *inth.suspend*.

The Inthreads code generation process [2] is divided into two stages. We extend this process in three steps as shown in figure 1.

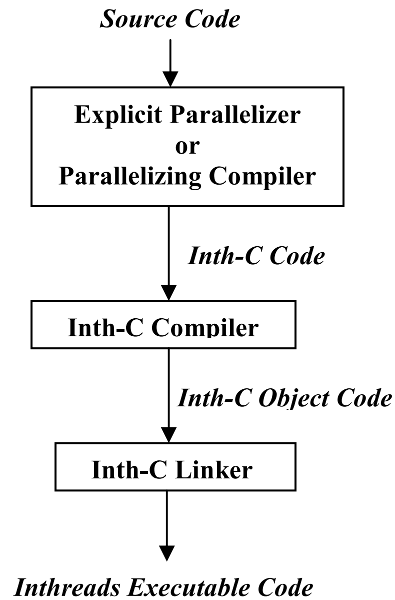

Fig. 1 Stages in Inthreads compilation

The parallelization step creates Inth-C code from given source code. The Inth-C compiler converts the parallel code into Inth-C object code (Inthreads compliant machine language). The Inth-C linker converts the Inth-C object code in to Inthreads executable code. For each thread there exists a parallel region, which is enclosed within a block of code marked with directive #pragma inthread. Following is an example of such code:

```
INTH_START (1, T1)
#pragma inthread
{
y = …..;
INTH_WAIT(1);
x+ = y;
}
return(x);
T1: # pragma inthread
{
x = …..;
INTH_SET(1);
INTH_HALT();
x+ = y;
}
```

Fig. 2 An Inth-C code

3.1 The ISB Architecture

Figure 3 shows the architecture of Instruction Stream Buffer (ISB):

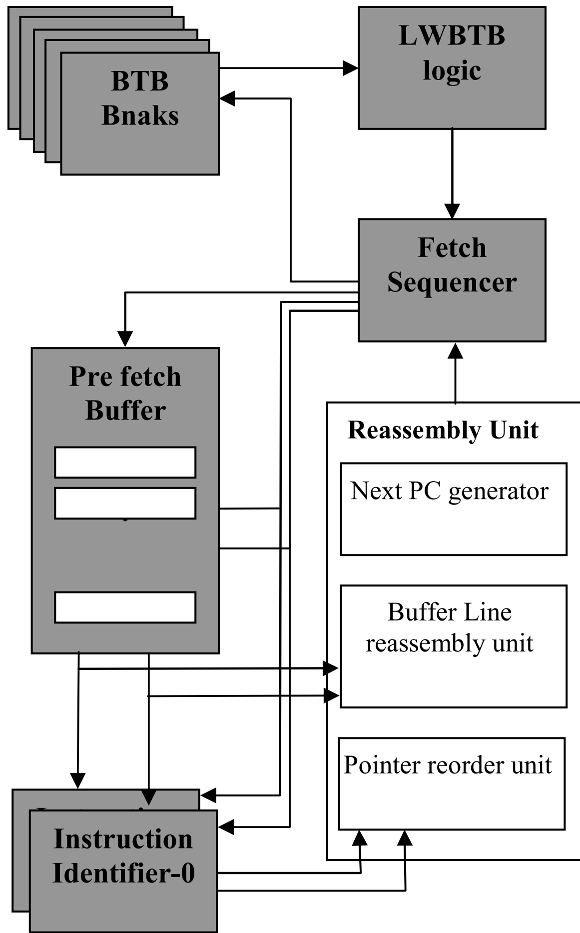

Fig. 3 The ISB Architecture

It consists of five key components namely: instruction identifier, fetch sequencer, prefetch buffer, LWBTB logic, and reassembly unit. The fetch sequencer directs the instructions for fetch direction as per the branch information maintained by it. The read address of LWBTB, prefetch buffer and instructions are generated by fetch sequencer. The ISB in most of current processors does not support instruction streaming beyond the basic blocks. The major components of reassembly units are: the next PC generator, the buffer line reassembly unit, and the pointer reorder unit. Two buffer lines can be reassembled by buffer line reassembly unit according to the design of ISB.

3.2 The Structure of ISB

In the design of LWBTB, the branch target buffer (BTB) can be redesigned to provide sufficient reassembling information for instruction stream buffer to have the reassembling ability. Figure 4 shows the structure of ISB illustrating an example of how the instruction stream is directed by fetch sequencer.

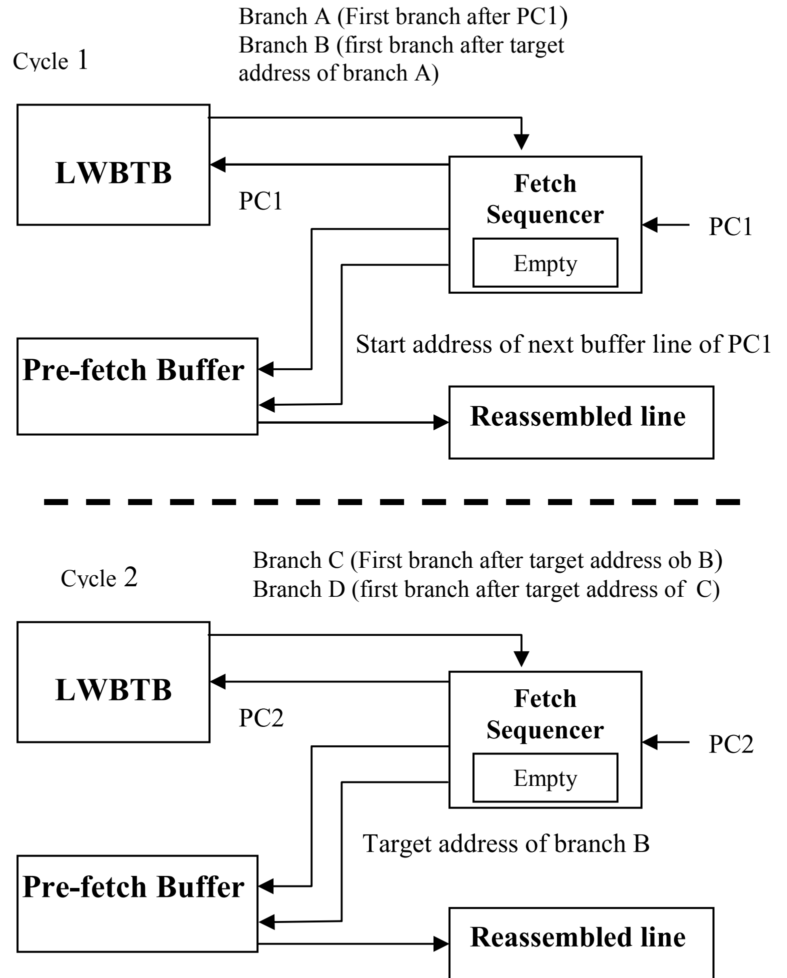

Fig. 4 The ISB structure

There is no branch information in fetch sequencer in cycle 1. Following two read addresses are supplied by the fetch sequencer to the prefetch buffer: (i) PC1, and (ii) the starting address of the next buffer line after PC1. PC1 is also accessed by fetch sequences to access LWBTB to get two branch information, say ‘A’ and ‘B’. If the branch ‘A’ is not the fetched instructions in cycle 1, it is stored in fetch sequencer, otherwise the information of branch ‘B’ is stored and the branch ‘A’ is assigned the next PC as target address. In cycle 2, it is assumed that branch ‘A’ is the fetched instruction in cycle 1 and the information of branch ‘B’ is stored in the fetch sequencer with PC2 as the target address of branch ‘A’.

4. Parallel code Compilation

The implicit assumption taken by conventional compilers, as the result of execution of threads in shared context, do not hold for Inthreads parallelized code. Let us consider the code given in section 3. the variable 'x' is assigned in thread T1 which is read later in main thread. It would not be aware of communication through 'x' separately if the compiler processed the code T12 independently, and it could erroneously identify assuming 'x' in T1 as dead code. The incorrect code generated by 'x' is then removed.

4.1 Internal Code Representation

Control flow graph (CFG) [15] is the best way for compilers to use internal code representation. The nodes of control flow graph represent basic blocks of the program and the edges represent the possibility of control flow path between various basic blocks. Control flow graph with information on parallel execution semantics is extended by concurrent control flow graph (CCFG) [5]. The CCFG introduces two additional edges which connect the outgoing edges of an INTH_START to the code of the starting thread. Second, the synchronization edges, which connect INTH_SET instruction to the corresponding INTH_WAIT in parallel threads. Figure 5 shows the CCFG of code given in section 3:

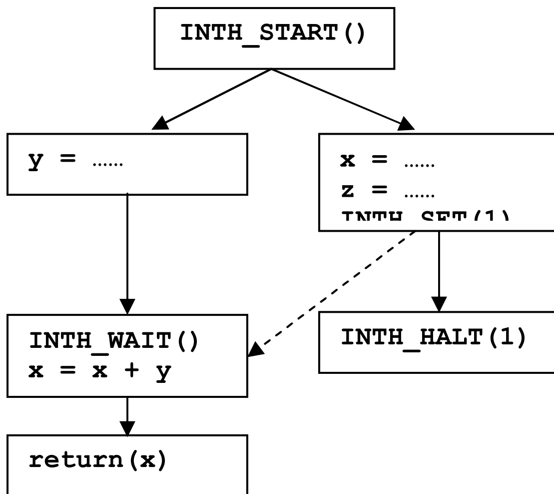

Fig. 5 CCFG of figure 2

4.2 Identification of Shared Variables

The detection of shared variables is discussed in [2]. The detection of shared variable conservatively considers all possible pairs of synchronized instructions as edges of flow graph.

4.3 Correctness Requirement for Compilers

There are mainly two stages in which compiler conceptually proceeds. The compiler can introduce new temporary variables, roughly corresponding to the set of transformation of compiler optimization during first stage. During the second stage, the compiler performs a mapping of statements from the set of program variables to the set location corresponding to register allocation. To show the correctness of compilation of context sharing multithreaded code [2], the generated data should be data-race-free. That means the execution is sequentially consistence and execution results of the compiled code are possible under the execution of the input source program.

4.4 Compilation optimization

In general, the optimization can be classified into data-flow-sensitive and data-flow-insensitive [15]. The data flow insensitive optimization has little interface with parallel execution of code but data flow sensitive optimization needs some significant adjustments. Dead code elimination tries to eliminate the code involving useless computation. A statement is said to be useless if it is never used in any of the following execution part. Common sub-expression elimination [1] eliminates duplicate steps of computation.

5. Mapping of Register Allocation

Register allocation has been uased by most of modern compilers. It is mapping of virtual registers to the limited set of architectural registers. Graph coloring [2] is the dominant approach for register allocation. It represents the problem by an inference graph. The inference graph uses nodes by representation of virtual registers and edges connect any two conflict nodes.

6. Implementation

Inth-C compiler is implemented for code compilation [2]. For this purpose a simulator based on SimpleScalar tool set [4] is used. A set of benchmarks is used for evaluation including benchmarks Spec2000 suite [11]. The benchmarks used from [11] are Mcf, Art, and Twolf, and remaining from Mediabench suite [3]. The following figure shows the effect of register file size on speedup achieved by comparison of benchmearks with original sequential versio

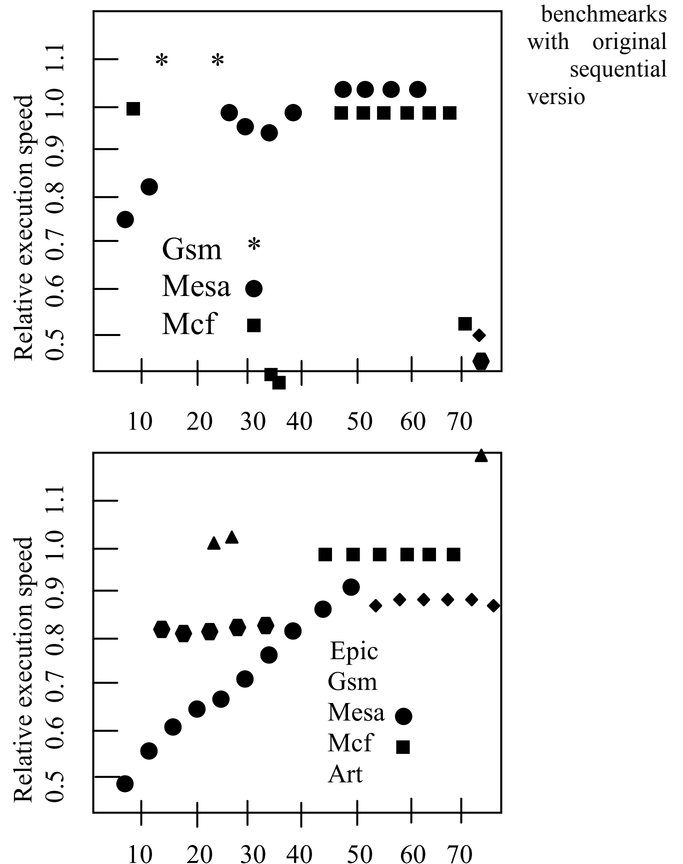

Fig. 6 Slow down of serial code as a function of register file size

It was observed that the result depends upon the register pressure in the original sequential code. For comparatively complex code like Epic, Art, and Mesa, the performance is improved significantly as per the number of registers. The remaining benchmarks are unchanged. When the effect of register file size on execution speed of original one is compared we find that only significantly affected sequential code is Mesa, which is slowed down by 30% while using twelve registers. The pressure is more prominent in parallel version where Mesa is slowed down by 50%.

7. CONCLUSION

In this paper we have shown that the integration of programming model with architecture benefits the hardware as well as the compiler. We have given a framework that performs the correctness analysis of compiler optimization. We have also applied a framework to prove correctness of compiler that supports register sharing architecture. The integration of compilation model with architecture can pave both sides. The limitations imposed on code can reduce the amount of conflict between the instruction of various threads and the hardware implementation can be simplified. Apart from this the consistency assurance supported by the architecture results in a simplified programming model.

REFERENCES

[1] Aho, A. V., R. Sethi, and J. D. Ullman, “Compilers. Principles, Techniques and Tools”, Addison Wesley, 2000.

[2] Alex G., Avi M. Assaf S., Gregory S., Code Compilation for an Explicitly Parallel Register-Sharing Architecture, IEEE International Conference on Parallel Processing, 2007

[3] C. Lee, M. Potkonjak, W. H. Mangoine-Smith, “Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems”, 30th Annual ACM International Symposium on Microarchitecture, 1997

[4] D. Burger, T. M. Tustin, S. Bennet, “Evaluating Future Microprocessors: The SimpleScalar Tool Set”, Technical Report CS-TR, University of Wisconsin Madison, 1996

[5] D. Grunwald, H. Srinivasan, “Data Flow Equations for Explicitly Parallel Programs”, Fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, 1993

[6] D. M. Tullsen, S. Eggers, H. M. Levy, “Simultaneous Multithreading: Maximizing on-chip Parallelism”, Proceeding of the 22th Annual International Symposium on Computer Architecture, 1995

[7] David I. August Wen-mei W. Hwu Scott A. Mahlke, The Partial Reverse If-Conversion Framework for Balancing Control Flow and Predication, International Journal of Parallel Programming Volume 27, Issue 5, Pages: 381 – 423, 1999

[8] Dionisios N. Pnevmatikatos Manoj Franklin, Control Flow Prediction for Dynamic ILP Processors, Proceedings of the 26th Annual International Symposium on Micro-architecture, 1993

[9] Guilin Chen, Mahmut Kandemir, Compiler-Directed Code Restructuring for Improving Performance of MPSoCs, IEEE Transactions on Parallel and Distributed Systems, Vol. 19, No. 9, 2008

[10] J. Cong, Guoling Han, Zhiru Zhang, “Architecture and compilation for data bandwidth improvement in configurable embedded processors”, IEEE International Conference on Computer Aided Design, Proceedings of the 2005

[11] J. L. Henning, “SPEC CPU 2000: Measuring CPU Performance in the new Millennium”, Computer 33(7), 2000

[12] L. Hammond, B. A. Nayfeh, K. Olukotun, “A single chip Multiprocessor”, IEEE Computer Special Issue on Billion-Transistor Processor, 30(9), 1997

[13] Marcos, Keali, “Exposing instruction level parallelism in the presence of loops”, Computation Systems Vol. 8 Number 1, pp. 074-085, 2004

[14] Noah Snavely, Saumya Debray, Gregory R. Andrews, Unpredication, Unscheduling, Unspeculation: Reverse Engineering Itanium Executable, IEEE Transactions on Software Engineering, Volume 31 Issue 2, 2005

[15] S. Muchnik, “Advanced compiler design and implementation”, Morgan Kaufmann Publishing, 1997

[16] Shaoshan Liu Gaudiot, J. L., The potential of fine-grained value prediction in enhancing the performance of modern parallel machines, Computer Systems Architecture Conference, 2008

[17] Steve Car, Combining Optimization for Cache and Instruction-Level Parallelism, Proceedings of PACT’96, 1996

[18] Siwei Shen, David Flanagan, Siu-Chung Cheung, “An Extended Heuristic for Hyper-block Selection in If-Conversion”, Department of Electrical Engineering and Computer Science University of Michigan, Ann Arbor, Michigan 48109, USA

[19] Su-Hui Chiang and Sangsuree Vasupongayya, Design and Potential Performance of Goal-Oriented Job Scheduling Policies for Parallel Computer Workloads, IEEE Transactions on Parallel and Distributed Systems, Vol. 19, no. 12, December 2008.

[20] Vijay S. Pai, Parthasarathy Ranganathan, Hazim Abdel-Shafi, and Sarita Adve, “The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors”, IEEE Transactions on Computers, Volume 48, Issue 2, Special issue on cache memory and related problems, pp 218-226, 1999