A New Compiler for Space-Time Scheduling of ILP Processors

Rajendra Kumar, *Member, IACSIT,* and P. K. Singh

***Abstract*—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 current 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 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. We want to achieve high level parallelism at faster clock speed it require distribution of processor resource and avoiding primitive that require single cycle global communication. Distribution of its resources, including instruction stream, register files, memory port and ALUs, over a pipelined two dimensional mesh interconnect are done by raw microprocessor [4]. In this paper, we propose a compiler RPCC for general purpose sequential programs on the raw machine.**

***Index terms*—ILP, Basic Block, benchmark, ISB**

I. Introduction

Instruction-level parallel processing (ILP) [3] 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 multi threading (SMT) [6] and chip multi processing (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].

There are three contribution in this paper, first we introduce Parallel Register Sharing Architecture, second on a raw machine we describe the space time scheduling of ILP processors on a raw machine by carrying some technique from the partitioning and scheduling of task on MIMD machine, third a new control flow model based on asynchronous local branches inside a machine with multiple independent instruction stream is introduced by it. Finally, it shows independent instruction machine giving the raw machine the ability to tolerate timing variation due to dynamic events.

II. 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 [11, 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 ILP processors.

Many commercially available embedded processors are

Manuscript received on February 22, 2011, accepted on April 01, 2011, and published in August 2010.

This work was supported in part by the Gautam Budh Technical University, Lucknow, India.

Rajendra Kumar is with Computer Science and engineering Department, Vidya College of Engineering, Meerut (Uttar Pradesh), India (phone: +91-9412002322, e-mail: rajendra04@gmail.com, website: http://www.rkronline.in).

P K Singh is with the Computer Science and engineering Department, MMM Engineering College, Gorakhpur (Uttar Pradesh), India (e-mail: topksingh@gmail.com).

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 presents 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 conFigd 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 [20]. 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.

III. The Architectural Motivations

We introduce Parallel Register Sharing Architecture and raw machine as one such machine.

A. The ISB Architecture

Fig 1 shows the architecture of Instruction Stream Buffer (ISB):

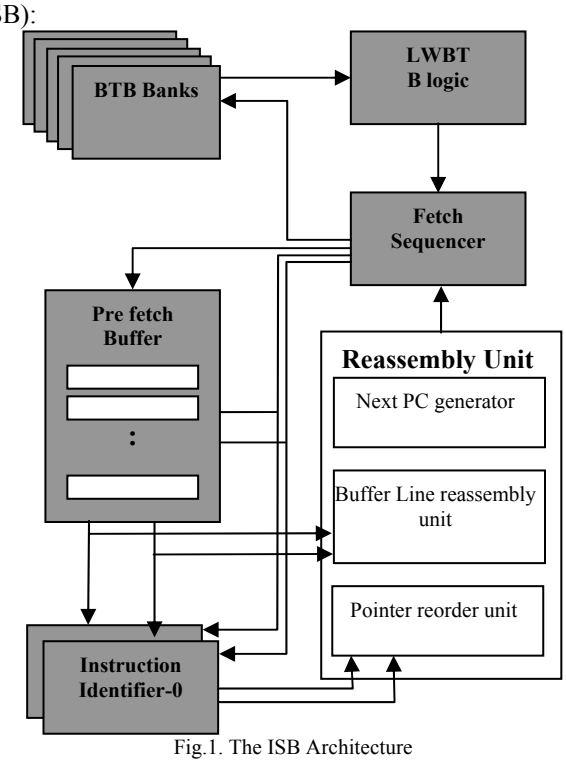


Fig.1. The ISB Architecture

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.

B. 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. Fig 2 shows the structure of ISB illustrating an example of how the instruction stream is directed by fetch sequencer.

There is no branch information in fetch sequencer in cycle 1. Following two read addresses are supplied by the fetch sequencer to the prefect 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*'.

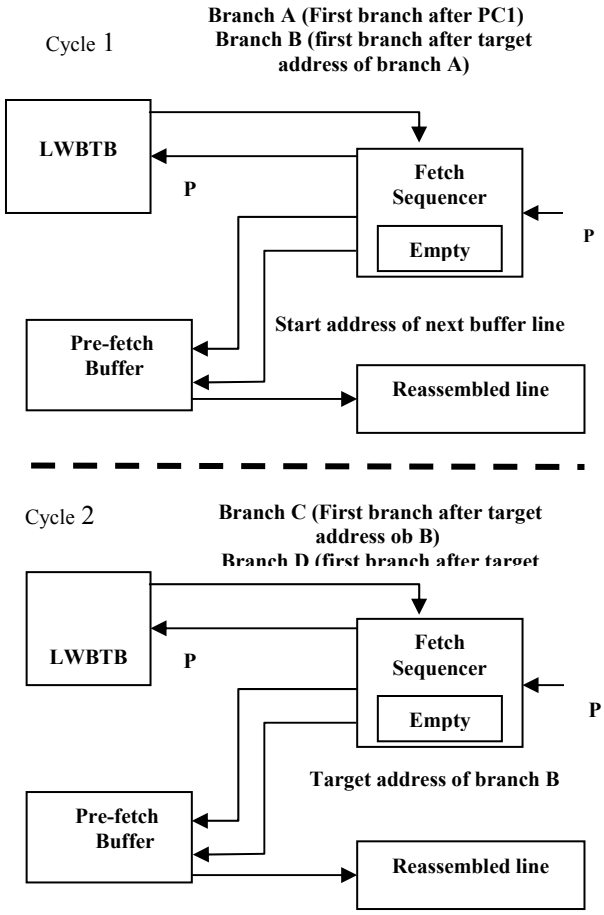


Fig. 2 The ISB structure

C. Raw Machine

Raw architecture inspire from VLIW machine they both share the common object of statically scheduling ILP. There are two differences between VLIW machine and raw machine. First, they differ in resource organization. VLIW machine of various degree of scalability which has been proposing ranging from completely centralized machine to machine with distributed functional unit that require file and memory. On the other hand, raw machine is the first ILP microprocessor that provides a software exposing scalability two dimensional interconnected between clusters of resources. This feature result in length of all wires to the distance between neighboring tiles, this enable in turns higher clock rating.

Second, the two machines also differ in their control flow model. VLIW machine has single flow of control while a raw machine is having multiple flow of control. This feature also increase available exploiting parallelism that enable asynchronous global branching and localization of control and improving tolerance of dynamic events.

D. Overview of space time scheduling

On a raw machine, the space time scheduling of ILP consist of orchestrating the parallelism within a basic block across the raw tiles and handling of the control flow across the basic block. It consists of several tasks the assignment of instruction to processing unit (spatial scheduling) the scheduling of that instruction on the tiles they are assigned (Temporal scheduling), the assignment of data to tiles, and the distinct orchestration of communication across a mesh interconnect both within and across the basic blocks

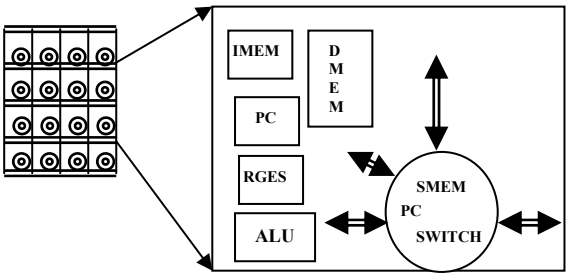


Fig. 3. Raw microprocessor in form of mesh of tiles

Compiler make the control flow between basic block better through asynchronous global branching, this mechanism uses the static network and individual branch on each tile, control localization allow some branch in the program.

Assignment and scheduling of instruction are the main task of the basic block. The assignment is done in three steps by the raw compiler. The clustering, merging and placement clustering group's together instruction that can usefully be utilized given the cost of communication. Merging reduces the number of cluster down to the no pf processing unit by the merging the cluster. Bijective mapping from the merged cluster to the PU is done by placement scheduler. Instruction scheduling is done with a traditional list scheduler. Data assignment and instruction assignment are implemented to allow flow of information in both direction, thus reflected the interdependent nature of the assignment problem. In unified manner, inter block and intra block communication are both identified and handled.

E. MIMD Task Scheduling

Assigning problem and scheduling instruction of compiler may be view in two ways: First is raw compiler statically scheduling ILP just like a VLIW compiler. So, a clustered VLIW with distributed register and functional unit face a similar problem as the raw machine. And, second is the compiler schedule task on a MIMD machine where task are at the granularity of instruction. At a coarser granularity, MIMD machine face a similar scheduling problem. MIMD scheduling research is applicable to clustered VLIW as well.

IV. The RPCC Compiler

Raw compiler, which compiles both *C* and *FORTRAN* programs, are implemented using the *SUIF* compiler infrastructure. It consist of three phase. High level programs analysis and transform perform in first phase, which contain map. The memory system is managed by raw compiler. The memory provided by maps and the data access model is briefly explained below. Initial phase also contain traditional technique such as memory disambiguation, loop unrolling and array reshape plus a new control optimization technique. Scheduling of ILP performs by space time scheduler in the second phase. In third phase compiler generate lode for the processor and the switches, with a few modification to handle the communication instruction and

the communication register.

Memory and data access model on a raw machine memory is distributed across the tiles. This model provides two ways of accessing this memory system, one for static reference and one for dynamic reference in static reference every invocation can be determined at compile-time to refer to memory on one specific tile. This property is also called the static residence property. The dynamic reference use the dynamic network to handled to necessary communication by disambiguating the address at run-time. Due to attraction of static reference raw compiler generate as many static references as possible. First reason is that, static reference can proceed without any of the overhead due to dynamic disambiguation and synchronization. And Second reason is that it can utilize the full memory bandwidth. We focus on results which can be attained when the raw compiler succeeds in identifying static references. Discussion of the compiler managed system can be found in [7].

Through intelligent data mapping and code transformation static reference can be created. For arrays, the raw compiler distributes them through low order interleaving, which interleaves the arrays element-wise across the memory system. To satisfy the static residence property, we have developed technique, which uses loop unrolling for array reference.

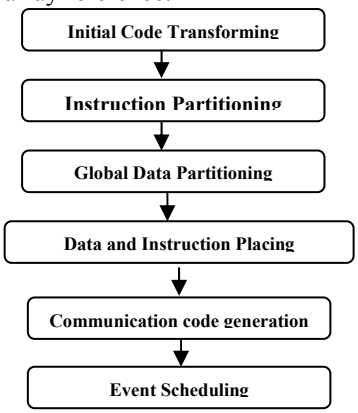


Fig. 4. Phases of RPCC compiler

Communication of Scalar values within basic blocks follows a dataflow model, so that the tile consuming a value receives it directly from the producer tile. Each program variable is assigned a home tile, to communicate values across basic block. The value of a variable is transferred from its home to the tiles, which use the variable at the beginning of a basic block. The value of a modified variable is transferred from the computing tile to its home tile at the end of a basic block. The phases of RPCC compiler are given by Fig 4.

The basic block organization exploits the ILP within a basic block by distributing the parallelism within the basic block across the tiles. It transforms a single basic block into an equivalent set of intercommunicating basic blocks. Orchestration consists of assignment and scheduling of instructions, assignment of data, how orchestration performs these functions.

Initial code transformation: The transformation is performed in this phase. And, remaining, phases convert statements of the basic block to static single assignment form removes anti-dependencies and Output-dependencies from the basic block it exposes available parallelism.

Second, two types of dummy instructions are used Read instructions are inserted for variables which are live-on-entry and read in the basic block. Write instructions are inserted for variables, which are live-on-exit and written within the basic block. Eventual representation of stitch code is simplified by this instruction. This representation allows the event scheduler to overlap the Stitch code with other in the basic block.

Third, the source program is decomposed into instructions in three-operand form. They correspond closely to the final machine instructions and their cost attributes can easily estimate. So, in atomic partitioning and scheduling units operands are logical candidates. In dependence graph, node represents an instruction, and an edge represents a true flow dependence between two instructions. Each node is labeled with the estimated cost of running the instruction.

Instruction practitioner: It divides the original instruction stream into multiple instruction streams, one for each tile. Instruction placer, which bind the resultant instruction streams to specific tiles. The partitioned attempts to balance the benefits of parallelism against the overheads instruction stream. Fig 5(b) shows a sample output of this phase.

Clustering: Clustering partitions instructions to minimize run-time, assuming non-zero communication cost is like as an idealized uniform network whose latency is the average latency of the actual network. The phase group's together instructions that either have no parallelism, or whose parallelism is too fine-grained to be utilized give the communication cost. RPCC applies a greedy technique based on the estimation of completion time called Dominant Sequent Clustering [26]. Initially, each instruction node belongs to a unit cluster. Topological order is used to visit the instruction node. It selects from the list of candidates the instruction on the longest execution at each step. It then checks whether the selected instruction can merge into the cluster of any of its parent instructions to reduce the estimated completion time of the program. Estimation of the completion time reflects the cost of both computation and communication. This algorithm completes when all nodes have been visited exactly once.

Merging: It combines clusters to reduce the number of clusters down to the number of tiles, maintain load balance and to minimize communication events are useful heuristic in merging. It is a locality-sensitive load balancing technique, which tries to minimize communication events used by compiler. The Raw compiler initializes *N* empty partitions and visits clusters in decreasing order of size. It merges the cluster into the partition with which it communicates the most, unless such merging results in a partition which is 20% larger than the size of an average partition when it visits a cluster. If the latter condition Occurs, the cluster is placed into the smallest partition instead.

Global data practitioner: Global data practitioner a scalar data element is assigned a "home" tile location to communicate values of data elements between basic blocks. Within basic blocks, the initial reads and the final write of a variable need to communicate with its home location. Dividing the task of data home assignment into data partitioning and data placement is done by raw complier

which like as instruction mapping.

To group data elements into sets, each of which is to be mapped to the same processor is the job of the data practitioner. RPCC performs global analysis to partition data element into sets which are accessed together. The step of algorithm for initialization, a virtual processor number is arbitrarily assigned to each instruction stream on each basic block, as well as to each scalar data element. Access patterns of each instruction stream are summarized with its affinity to each data element. Affinity means for a data element if it either accesses the element, or it produces the final value for the element in that basic block. It remaps instruction streams to virtualized processors given fixed mapping of data elements after first initialization. Only true data element is remapped in this phase. This process repeats until no improvement of locality can be found. Data elements mapped to the same virtual processor are likely related based on the access patterns of the instruction streams in resulting partition.

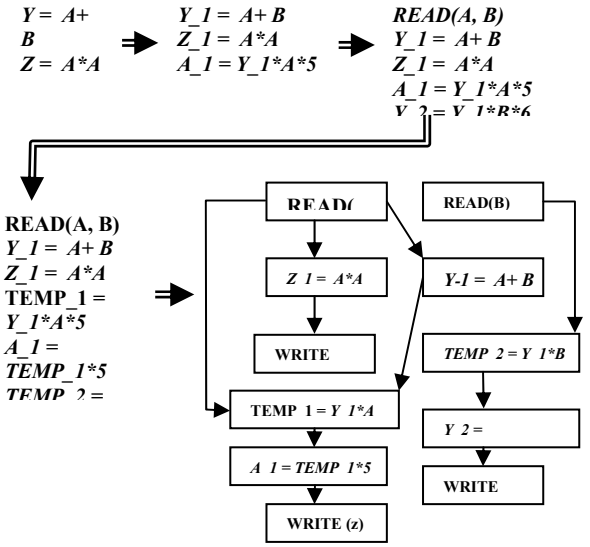
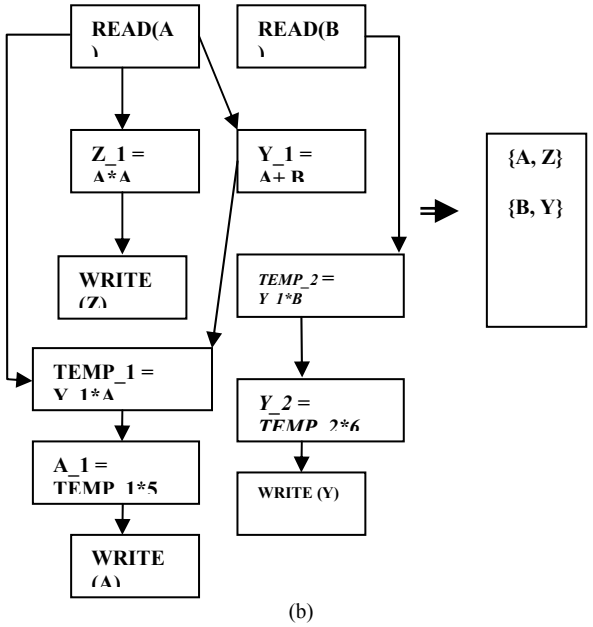


Fig. 5(a) The initial program undergoing transformations made by initial code transformation;



(b)

Fig. 5(b) Shows result of instruction partitioning, (c) shows result of global data partitioning

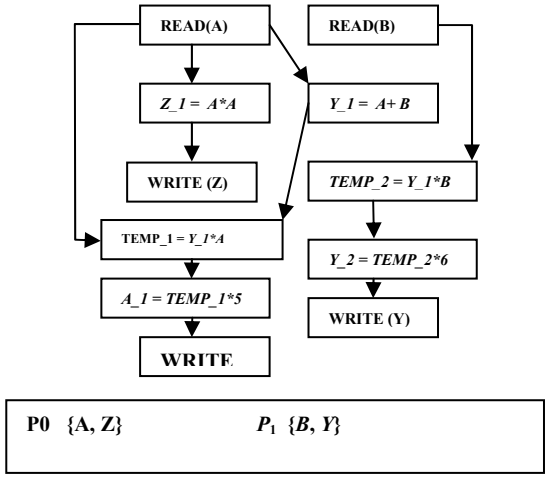


Fig. 5(d) Result of data and instruction placer

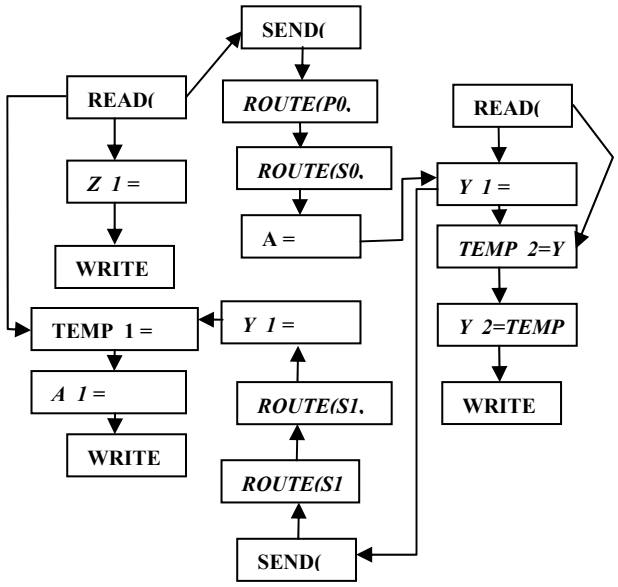


Fig. 5(e) Result of communication code generator

Data and Instruction placer: Instruction placer mapping of virtualized data sets and instruction streams to physical processors is done by data and instruction placer. Fig 3d shows a sample output of this phase. The placement phase removes the assumption of the idealized interconnects and takes into account the non-uniform network latency. Placement of each data partition is currently driven by those data elements with processor preferences, i.e., those corresponding to fixed memory references. All these function performed before instruction placement to allow cost estimation during placement to account for the location of data. RPCC uses a swap-based greedy algorithm to minimize the communication bandwidth. It is initially for instruction placement. It assigns clusters to tiles, and looks for pairs of mappings that can be swapped to reduce the total number of communication hops.

Communication code generation: Communication code generation in RPCC uses dimension-ordered routing; this spatial aspect of communication scheduling is completely manual. If contention is determined to be a performance bottleneck, a more flexible technique can be employed.

Event scheduler: To provide the minimal estimated run-time, event scheduler schedules the computation and communication events within a basic block. Routing is done in raw compiler itself. The scheduling problem is a

generalization of the traditional instruction scheduling problem. Responsibility of instruction is avoid deadlocks in the network the Raw compiler can avoid the need for such management by using single source unit. When communication path is as a single scheduling unit. When a communication path is scheduled, the path incurs no delay in the static scheduling.

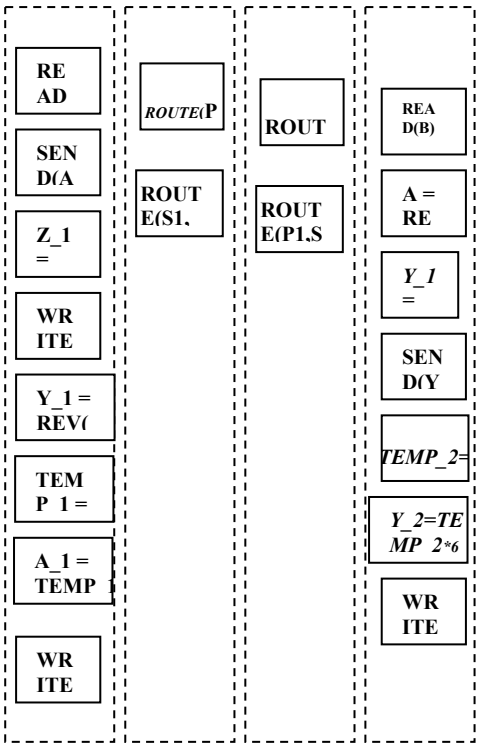


Fig. 5 (f) Shows final result after event scheduler.

Event scheduling is a static problem. It remains deadlock-free and correct even in the presence of dynamic events such as cache misses. The static ordering property which allows the schedule to be stored as compact instruction streams. Instruction stream do not require timing information to ensure correctness. Fig 5(f) shows a sample output of the event scheduler. First proper ordering of the route instructions on the switches, and, second, the successful overlap of computation with communication on *WYX*, where the processor computes and writes *Z* while waiting on the value of output.

RPCC uses a single greedy list scheduler to schedule both computation and communication. This algorithm keeps track of a ready list of tasks. A task is either a computation or a communication path it selects and schedules the task on the ready list with the highest priority. Scheme is based on the following observation:

The priority of a task should be directly proportional to the impact it has on the completion time of the program. This impact, in turn, is lower-bounded by two properties of the task: its level, defined to be its critical path length to an exit node; and its average fertility, defined to be the number of descendent nodes divided by the number of processors. Therefore, we define the priority of a task to be a weighted sum of these to properties.

V. Control Orchestration

To exploit ILP within a basic block raw tiles are cooperated. The Raw compiler has to orchestrate the control flow on all the tiles between basic blocks. This orchestration is performed through asynchronous global branching and control optimization which localizes the effects of a branch in a program to a single tile.

Asynchronous global branching by using the static network and local branches, raw machine implement global branching asynchronously in software. Static network broadcast branch value to all the tiles. All these function are scheduled by the compiler. So that it can overlap with other computation in the basic block. Each tile and switch individually performs a branch without synchronization at the end of its basic block execution.

VI. Control Localization

Control localization is the technique of treating a branch-containing code sequence as a single unit during assignment and scheduling. This scheduling unit is called a macro-instruction.

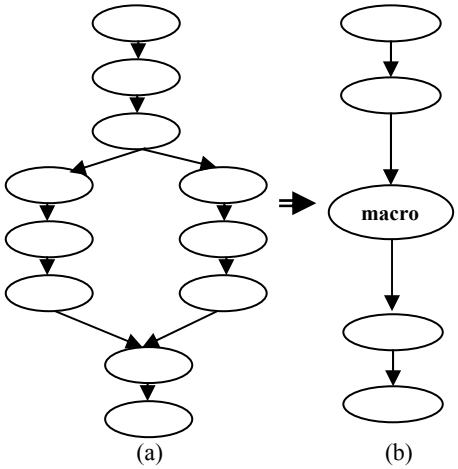


Fig. 4. An illustration of control localization. (a) control flow graph before control localization. (b) shows the control flow graph after control localization.

The technique which allows a raw machine to execute different macro-instructions concurrently on different tiles is a control optimization made possible though raw s independent flows of control, control localization avoids the broadcast cost of asynchronous raw compiler following steps to control localize a code sequence, it verifies that the code sequence can in fact be placed on a single tile, which means that either (1) all its memory operations refer to a single tile, or (2) enough memory operations and all their preceding computations can safely be separated from the code sequence so that (1) is satisfied.

Next, the compiler identifies the input variables the code sequence requires and the output variables the code sequence generates. These variables are computed by taking the union of their corresponding sets over all possible paths within the code sequence. In addition, given a variable for which a value is generated on one but not all paths of the program, the variable has to be considered as an input variable as well. This input is needed to allow the code sequence to produce a valid value of the variable independent of the path traversed inside it. The result of identifying these variables is that the code sequence can be

assigned and scheduled like a regular instruction during basic block orchestration.

VII. 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].

VIII. 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. Register allocation has been used 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.

IX. Result

We measure the portion of the performance due to high level transformations and advanced locality optimizations and show the whole performance of the compiler. Experiments are performed on the raw simulator, which simulates the raw prototype. Latencies of the basic instructions are as follows: 2-cycle load, 1-cycle store, 1-cycle integer add or subtract; 12-cycle integer multiply, 35-cycle integer divide; 2-cycle floating add or subtract; 4-cycle floating multiply, and 12-cycle floating divide. Table 1 gives basic characteristics of the benchmarks.

TABLE 1. Benchmark Characteristics

Name of Bench-mark	Source	Languages	Number of lines of code	Size of primary array	Sequential RT (Cycles)
Fppp kernal	Spec92	FORTRAN	735	-	8.98 K
btrix	Spec92	FORTRAN	238	15x15x15x5	288 M
cholesky	Spec92	FORTRAN	128	3x32x32	34.4 M
vpentra	Spec92	FORTRAN	158	32x32	21 M
tomcatv	Spec92	FORTRAN	254	32x32	79 M
mxm	rawbench	FORTRAN	64	32x64, and 32x8	2 M
life	rawbench	C	118	32x32	2.45 M

Table 2 shows the speedups attained by the benchmarks for raw machines of various numbers of tiles.

TABLE 2. Benchmark Speedup

Name of Benchmark	$N = 2^0$	$N = 2^1$	$N = 2^2$	$N = 2^3$	$N = 2^4$	$N = 2^5$
Fppp kernal	0.49	0.68	1.35	3.01	6.03	9.40
btrix	0.83	1.47	2.60	4.41	8.58	9.65
cholesky	0.89	1.68	3.39	5.48	10.31	14.90
vpentra	0.70	1.77	3.32	6.37	10.60	19.34
tomcatv	0.93	1.65	2.78	5.53	9.90	19.45
mxm	0.95	1.98	3.61	6.64	12.21	23.56
life	0.94	1.70	3.01	6.64	12.66	23.54

All the benchmarks except fppp-kernel are dense matrix applications. These applications perform particularly well on a Raw Machine because by unrolling the loop. Raw compiler unrolls loops by the minimum amount required to guarantee the static residence property. The btrix benchmark can not unroll tie loop because its inner loop handle array dimensions of either five or fifteen. Sic block orchestrate is at most five or fifteen.

The fppp-kernel is different from the rest of the application in that it continue irregular fine grain parallelism. The benchmark fppp kernel on single tile the code generated by the Raw compiler is worse than that generated by the original MIPS compiler. The reason is that the raw compiler attempts to expose the maximal amount of parallelism without regard to register pressure. As the number of tiles increases, however, the number of available registers increases correspondingly, and the spill penalty of this instruction scheduling policy reduces.

X. 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.

We have also described how to compile a sequential program by a next generation processor that has asynchronous, physically distributed hardware that is fully-exposed to the compiler. The compiler partitions and schedules the program so as to best utilize the hardware. Together, they allow applications to use instruction-level parallelism to achieve high levels of performance. We have introduced the resource allocation approach of the raw compiler, which are based on MIMD task clustering and merging techniques global branching sequential control flow. Finally we have presented performance results which

demonstrate that for a number of sequential benchmark codes.

REFERENCES

[1] Aho, A. V., R. Sethi, 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] D. August, W. Hwu, and S. Mahlke, "A Framework for Balancing Control Flow and Predication", In Proceedings of the 30th InternationalSymposium on Microarchitecture, Dec. 1997.

[4] D. Burger, T. M. Tustin, S. Bennet, "Evaluating Future Microprocessors: The Simple Scalar 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

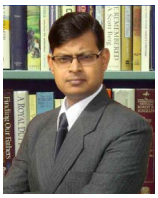
[16] Shaoshan Liu Gaudiot, J. L., The potential of fine-grained value prediction in enhancing the performance of modern parallel machines, Computer Systems Architecture Conference, 2008

[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] www.trimaran.org



Rajendra Kumar, btained B. E. degree from BIET Jhansi, M. Tech. from UPTU Lucknow, in Computer Science & Engineering. His area of interest includes Theoretical Computer Science, Human Computer Interaction, Computer Graphics, Biometric Systems, ICT, etc. His current research area is Instruction Level Parallelism. He is Associate Professor and Head of Computer Science & Engineering department at Vidya College of engineering, Meerut. He is active reviewer of Journal of Computational Biology and Bioinformatics Research, Nairobi. He is author of four text books including Theory of Automata, Languages & Computation from McGraw Hill, and eight distance learning books for MDU Rohtak, CDLU Sirsa, MGU Kerla, etc.

Prof. Kumar is member of IACSIT Singapore, CSTA USA, IAENG Hong Kong, ISTE New Delhi and Amnesty International UK. Prof. Kumar also has been Member of Board of Studies of Uttar Pradesh Technical University, Lucknow.

P K Singh, graduated from M M M Engineering College, Gorakhpur with the Bachelor of Computer Science degree and M.Tech. from University of Roorkee (now IIT Roorkee) in Computer Science and Technology then obtained a Doctor degree in the area of Parallelizing Compilers. He teaches a number of Computer Science subjects including Compiler Design, Automata Theory, Advanced Computer Architectures, Parallel Computing, Data Structures and Algorithms, Object Oriented Programming C++ and Computer Graphics etc., but mostly he teaches Compiler Design and Parallel Computing.

He is an Associate Professor of Computer Science & Engineering at MMM Engineering College, Gorakhpur.

Prof. Singh is author of Computer graphics book. He is member of several sicieties and bodies of Different organizations and universities.