International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

The process of Optimal Code Generation and its implementation in Parallel Processors

A. Alekhya, M.Tech.(CSE)1, Prof. Dr. G. Manoj Someswar, Ph.D.[2]

[1]*Research Scholar at Pacific University, Udaipur, Rajasthan, India.*
[2]*Professor & Research Supervisor, Pacific University, Udaipur, Rajasthan, India.*

***Abstract*-- The process of code generation is chiefly involved with three interrelated optimization tasks: instruction selection (with resource allocation), instruction scheduling and register allocation. For most of the architectures and in most situations, these tasks have been discovered to be NP-hard. A common approach for code generation includes solving each task separately, i.e. in a decoupled manner, which is easier from a software-engineering point of view. Phase-decoupled compilers generate good code quality for regular architectures, but if applied to DSPs the generated code results in with significantly lower performance due to strong interdependences between the different tasks. Generating code for DSPs drives to afford spending considerable resources in time and space on optimizations. Generating efficient code for irregular architectures requires an integrated method that optimizes simultaneously for instruction selection, instruction scheduling, and register allocation. Moreover, embedded systems have turned a prevalent part of our dialy life and this trend is very unlikely to decline anytime soon. Traditional superscalar techniques require for a 2–3× speedup in performance very roughly about an increase of 80× in area and, maybe even more important, about 12× in power consumption. For numerous mobile applications with critical energy and cost requirements, this is a cost too high to bear. As such, the generation of code for parallel processing systems has been, and remains, an important area of research. The principal aim of this thesis is to explore the processes of optimal code generation and how they are implemented in parallel processors besides elaborating the challenges and possible solutions in this arena.**

***Keywords*-- optimal instruction scheduling, semiconductor technology, parallel processing, code generation framework, ILP architecture**

I. INTRODUCTION

The computer industry has grown accustomed to, and has come to take for granted, a spectacular rate of increase in microprocessor performance, all of this without requiring a fundamental rewriting of the program in a parallel form, without using a different algorithm or language, and often without even recompiling the program.

Higher levels of performance benefit from improvements in semiconductor technology which permit shorter gate delays and higher levels of integration, both of which enable the construction of faster computer systems. Further speedups must come, primarily, from the use of some form of parallelism.

Instruction-level parallelism

Instruction-level parallelism results from a set of processor and compiler design techniques that speed up execution by causing individual RISC-style machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel. ILP systems are given a conventional high-level language program written for sequential processors and use compiler technology and hardware to automatically exploit program parallelism. Thus an important feature of these techniques is that like circuit speed improvements, but unlike traditional multiprocessor parallelism and massively parallel processing, they are largely transparent to application programmers. In the long run, it is clear that the multiprocessor style of parallel processing will be an important technology for the main stream computer industry. For the present, instruction-level parallel processing has established itself as the only viable approach for achieving the goal of providing continuously increasing performance without having to fundamentally re-write applications.

II. MOTIVATION

There has been substantial progress in the development of new methods in code generation for scalar and instruction-level parallel processor architectures during the last two decades. New retargetable tools for instruction selection have emerged, such as IBURG [FHP92, FH95]. New methods for fine-grain parallel loop scheduling have been developed, such as software pipelining [AN88, Lam88].

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

Global scheduling methods like trace scheduling [Fis81, Ell85], percolation scheduling [Nic84, EN89], or region scheduling [GS90] allow to move instructions across basic block boundaries. Also, techniques for speculative or predicated execution of conditional branches have been developed [HHG+ 95]. Finally, high-level global code optimization techniques based on data flow frameworks, such as code motion, have been described in [KRS98].

Most of the important optimization problems in code generation have been found to be NP-complete. Hence, these problems are generally solved by heuristics. Global register allocation is NP-complete, as it is isomorphic to coloring a live-range interference graph [CAC+ 81, Ers71] with a minimum number of colors. Time-optimal instruction scheduling for basic blocks is NP-complete for almost any nontrivial target architecture [AJU77, BRG89, HG83, MPSR95, PS93] except for certain combinations of very simple tar- get processor architectures and tree-shaped dependency structures [BGS93, BG89,BJPR85,EGS95,Hu61,KPF95,MD94,PF91]. Space-optimal instruction scheduling for DAGs is NP-complete [BS76, Set75], except for tree-shaped [BGS93, SU70] or series-parallel [Güt81] dependency structures. Instruction selection for basic blocks with a DAG-shaped data dependency structure is assumed to be NP-complete, too, and the dynamic programming algorithm designed for (IR) trees can no longer guarantee optimality for DAGs, especially in the presence of non-homogeneous register sets [Ert99].

Optimal selection of spill candidates and optimal a-posteriori insertion of spill code for a given fixed instruction sequence and a given number of avail- able registers is NP-complete even for basic blocks and has been solved by dynamic programming or integer linear programming for various special cases of processor architecture and dependency structure [AG01, HKMW66, HFG89, MD99].

For the general case of DAG-structured dependences, various algorithms for time-optimal local instruction scheduling have been proposed, based on integer linear programming e.g. [GE92, Käs00a, WLH00, Zha96], branch-and-bound [CC95, HD98, YWL89], and constraint logic programming [BL99]. Dynamic programming has been used for time-optimal [Veg92] and space- optimal [Kes98] local instruction scheduling.

The task of generating target code from an intermediate program representation can be mainly decomposed into the interdependent sub-problems of instruction selection, instruction scheduling, and register allocation. These sub-problems span a three-dimensional problem space. Phase-decoupled code generators proceed along the edges of the cube, while an integrated solution directly follows the diagonal, considering all sub-problems simultaneously.

The first criterion of a compiler is to produce correct code, but often correct code is not sufficient. Users expect programs to use available hardware re- sources efficiently. The current state-of-the-art in writing highly optimized applications for irregular architectures offers two alternatives. First, writing the applications directly in the assembly code for the specific hardware. Often, it is possible to automatically generate assembly code for those parts of the program that are not critical for the application, and only concentrate on the computationally expensive parts and write code for them by hand. Next, obtain highly optimized libraries from the hardware provider for a given target architecture. Then, the work consists in identifying parts of the application that may use a given library and call it. However, there are few companies that can spend sufficient effort in implementing highly optimized general purpose libraries that are as well handwritten directly in the assembly language by experts. Designing a good library may be a difficult task, and it may work only for specific application areas.

With the solutions above it is possible to generate highly optimized code but at high cost in terms of man months. Further, the code needs to be rewritten almost completely if the underlying hardware changes, since the methods are rarely portable. In terms of software engineering, maintainability is difficult as long as there is no unified framework where third-party components can be updated more frequently than the application itself. Within this work we aim at improving the current state-of-the-art in compiler generation for irregular architectures, where the user keeps writing applications in a high level language and the compiler produces highly optimized code that exploits hardware resources efficiently. We focus particularly on code generation and aim at producing an optimal code sequence for a given IR of an input program. Optimal code sequence is of particular interest for fixed applications where the requirements are difficult to meet, and often a dedicated hardware is required. Furthermore, from the hardware designer viewpoint, information about optimal code sequence may influence further design decisions.

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

Moreover, we consider the retargetability issue. The code generation framework should not encapsulate target hardware specific, but take the hardware description as input information together with the application specified in a high level language. From the hardware description it is either possible to generate a compiler (code generator generator), or parameterize an existing framework. A code generator is usually more difficult to build, thus in this thesis we implement a prototype of a parameterizable retargetable framework for irregular architectures.

The advances in the processing speed of current microprocessors are caused not only by progress in higher integration of silicon components, but also by exploiting an increasing degree of instruction-level parallelism in programs, technically realized in the form of deeper pipelines, more functional units, and a higher instruction dispatch rate. Generating efficient code for such processors is largely the job of the programmer or the compiler back-end. Even though most superscalar processors can, within a very narrow window of a few subsequent instructions in the code, analyze data dependences at runtime, reorder instructions, or rename registers internally, efficiency still depends on a suitable code sequence.

III. Research Question

The present attempt is going to deal with the processes of optimal code generation and their implementation in parallel processors. To begin with, it provides an overview of architecture of parallel processors. Next it turns its focus to code generation for parallel processors and how to optimize the process of code generation. So the main objectives of this study are-

1. To provide an overview of architecture of parallel processors.
2. To discuss various compiler design techniques that speed up execution by causing individual machine operations to execute in parallel.
3. To evaluate the performance indicators of parallel processors in comparison to other processors.
4. To analyze the applications of parallel processors and their impacts and implementation on computing world.
5. To describe the processes of code generation and optimization techniques from operations research.
6. To develop a framework for integrated code generation with algorithms to optimally solve the main tasks of code generation
7. To address the problem of overlapping register classes as a future work.

IV. Research Methodology

1 To utilize graphical tools for developing the architecture of parallel processors
2 To utilize various mathematical tools for optimizing code generation techniques
3 Determining how the research is going to be conducted
4 Collection of the research data
5 Analysis and interpretation of the research data
6 Writing up of the research paper.

V. Limitations Of The Study

Code generation consists mainly of three interrelated optimization tasks: instruction selection (with resource allocation), instruction scheduling and register allocation. These tasks have been discovered to be NP-hard for most architecture and most situations. A common approach to code generation consists in solving each task separately, i.e. in a decoupled manner, which is easier from a software engineering point of view.

In this research paper, we are going to develop a framework for integrated code generation with algorithms to optimally solve the main tasks of code generation in a single and fully integrated optimization step for regular and irregular architectures, using dynamic programming and integer linear programming.

We first provide the concept of time profile and the compression theorem for regular architectures. The dynamic programming algorithm for superscalar and regular VLIW processors is suitable for small and medium-sized problem instances. Secondly, for irregular architectures we introduced the concept of space profiles to describe data locality information and provided the compression theorem for irregular architectures. The dynamic programming algorithm for clustered VLIW processors is applicable to small but not trivial problem instances.

The dynamic programming method is suitable for optimizing for time, space, energy, and mixed goals. We adopt an energy model from the literature and presented a framework for energy-optimal integrated local code generation. We define a suitable power profile, which is the key to considerable compression of the solution space in our dynamic programming algorithm. Our method is generic and not limited to a fixed power model. If more influence factors are to be considered that are known at compile time, it can easily be adapted by modifying the power profile definition accordingly.

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

In order to address larger problem instances we describe an optimization technique that exploits partial symmetries for regular architectures. The idea of pruning partial solutions that can be shown to be equivalent to others by analyzing local symmetries in scheduling situations did not lead to substantial improvements because the (moderate) size reduction of the solution space was outweighed by the symmetry analysis time. Partial-symmetry optimization techniques still need further investigations and extensions for irregular architectures.

To begin with, this research paper gives an overview of instruction-level parallelism. Under theoritical background, it interprets architecture of parallel processors. Then it explores and proposes various compiler design techniques. It analyzes and critically evaluates performance of parallel processors. Next it explores significant applicatoins of parallel processors. Code generation and optimization techniques are proposed and discussed in detail. Finally it concludes with integrated code generation techniques.

VI. ARCHITECTURE OF PARALLEL PROCESSORS

Computer architecture is a contract between the class of programs that are written for the architecture and the set of processor implementations of that architecture. Usually this contract is concerned with the instruction format and the interpretation of the bits that constitute an instruction, but in the case of ILP architectures it extends to information embedded in the program pertaining to the available parallelism between the instructions or operations in the program.

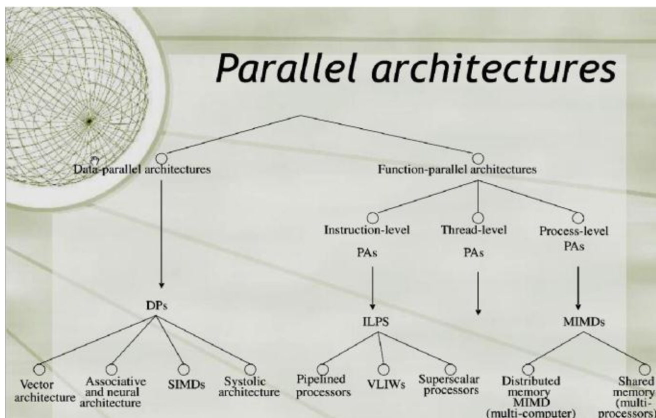

Figure 1 showing Parallel Architectures

ILP architectures can be classified as:

Sequential architectures: architectures for which the program is not expected to convey any explicit information regarding parallelism. Superscalar processors are representative of ILP processor implementations for sequential architectures.

Dependence architectures: architectures for which the program explicitly indicates the dependences that exist between operations. Dataflow processors ([29-31]) are representative of this class.

Independence architectures: architectures for which the program provides information as to which operations are independent of one another. Very Long Instruction Word (VLIW) processors are examples of the class of independence architectures.

If ILP is to be achieved, between the compiler and the runtime hardware, the following functions must be performed: the dependences between operations must be determined,the operations, that are independent of any operation that has not as yet completed, must be determined, and these independent operations must be scheduled to execute at some particular time, on some specific functional unit, and must be assigned a register into which the result may be deposited.

Sequential Architectures and Superscalar Processors

The program for a sequential architecture contains no explicit information regarding the dependences that exist between instructions. Consequently, the compiler need neither identify parallelism nor make scheduling decisions since there is no explicit way to communicate this information to the hardware. (It is true, nevertheless, that there is value in the compiler performing these functions and ordering the instructions so as to facilitate the hardware's task of extracting parallelism.) In any event, if instruction-level parallelism is to be employed, the dependences that exist between instructions must be determined by the hardware. It is only necessary to determine dependences with sequentially preceding operations that are in flight, i.e., those that have been issued but have not yet completed. The operation is now independent of all other operations and may begin execution. At this point, the hardware must make the scheduling decision of when and where this operation is to execute.

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

A superscalar processor strives to issue an instruction every cycle, so as to execute many instructions in parallel, even though the hardware is handed a sequential program. The problem is that a sequential program is constructed with the assumption only that it will execute correctly when each instruction waits for the previous one to finish, and that is the only order that the architecture guarantees to be correct. The first task, then, for a superscalar processor is to understand, for each instruction, which other instructions it actually is dependent upon. With every instruction that a superscalar processor issues, it must check whether the instruction's operands (registers or memory locations that the instruction uses or modifies) interfere with the operands of any other instruction in flight, i.e., one that is either: already in execution, or has been issued but is waiting for the completion of interfering instructions that would have been executed earlier in a sequential execution of the program.

If either of these conditions is true, the instruction in question must be delayed until the instructions on which it is dependent have completed execution. For each waiting operation, these dependences must be monitored to determine the point in time at which neither condition is true. When this happens, the instruction is independent of all other uncompleted instructions and can be allowed to begin executing at any time thereafter. In the meantime, the processor may begin execution of subsequent instructions which prove to be independent of all sequentially preceeding instructions in flight. Once an instruction is independent of all other ones in flight, the hardware must also decide exactly when and on which available functional unit to execute the instruction. The Control Data CDC 6600 employed a mechanism, called the scoreboard, to perform these functions. The IBM System/360 Model 91, built in the early 1960s, utilized an even more sophisticated method known as Tomasulo's Algorithm to carry out these functions.

The further goal of a superscalar processor is to issue multiple instructions every cycle. The most problematic aspect of so doing is that of determining the dependences between the operations that one wishes to issue simultaneously. Since the semantics of the program, and in particular the essential dependences, are specified by the sequential ordering of the operations, the operations must be processed in this order to determine the essential dependences. This constitutes an unacceptable performance bottleneck in a machine that is attempting parallel execution. On the other hand, eliminating this bottleneck can be very expensive as is always the case when attempting to execute an inherently sequential task in parallel.

Independence architectures and VLIW processors

In order to execute operations in parallel, the system must determine that the operations are independent of one another. Superscalar processors and dataflow processors represent two ways of deriving this information at run-time. In the case of the dataflow processor, the explicitly provided dependence information is used to determine when an instruction may be executed so that it is independent of all other concurrently executing instructions. The superscalar processor must do the same but, since programs for it lack any explicit information, it must also first determine the dependences between instructions. In contrast, for independence architecture, the compiler identifies the parallelism in the program and communicates it to the hardware by specifying which operations are independent of one another. This information is of direct value to the hardware, since it knows with no further checking which operations it can execute in the same cycle. Unfortunately, for any given operation, the number of operations of which it is independent is far greater than the number of operations on which it is dependent. So, it is impractical to specify all independences. Instead, for each operation, independences with only a subset of all independent operations (those operations that the compiler thinks are the best candidates to execute concurrently) are specified.

By listing operations that could be executed simultaneously, code for an independence architecture may be very close to the record of execution produced by an implementation of that architecture. If the architecture additionally requires that programs specify where (on which functional unit) and when (in which cycle) the operations are executed, then the hardware makes no run-time decisions at all and the code is virtually identical to the desired record of execution.

The VLIW processors that have been built to date are of this type and represent the predominant examples of machines with independence architectures. The program for a VLIW processor specifies exactly which functional unit each operation should be executed on and exactly when each operation should be issued so as to be independent of all operations that are being issued at the same time as well as of those that are in execution.

A particular processor implementation of VLIW architecture could choose to disregard the scheduling decisions embedded in the program, making them at run-time instead. In doing so, the processor would still benefit from the independence information but would have to perform all of the scheduling tasks of a superscalar processor.

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

Furthermore, when attempting to execute concurrently two operations that the program did not specify as being independent of each other, it must determine independence, just as a superscalar processor must.

With a VLIW processor, it is important to distinguish between an instruction and an operation. An operation is a unit of computation, such as an addition, memory load or branch, which would be referred to as an instruction in the context of a sequential architecture. A VLIW instruction is the set of operations that are intended to be issued simultaneously. It is the task of the compiler to decide which operations should go into each instruction. This process is termed scheduling.

Conceptually, the compiler schedules a program by emulating at compile-time what a dataflow processor, with the same execution hardware, would do at run-time. All operations that are supposed to begin at the same time are packaged into a single VLIW instruction.

The order of the operations within the instruction specifies the functional unit on which each operation is to execute. A VLIW program is a transliteration of a desired record of execution which is feasible in the context of the given execution hardware.

The compiler for a VLIW machine specifies that an operation be executed speculatively merely by performing speculative code motion, that is, scheduling an operation before the branch that determines that it should, in fact, be executed. At run-time, the VLIW processor blindly executes this operation exactly as specified by the program just as it would for a non-speculative operation. Speculative execution is virtually transparent to the VLIW processor and requires little additional hardware. When the compiler decides to schedule an operation for speculative execution, it can arrange to leave behind enough of the state of the computation to assure correct results when the flow of the program requires that the operation be ignored. The hardware required for the support of speculative code motion consists of having some extra registers, of fetching some extra instructions, and of suppressing the generation of spurious error conditions.

The VLIW compiler must perform many of the same functions that a superscalar processor performs at run-time to support speculative execution, but it does so at compile-time. The earliest VLIW processors built were the so-called attached array processors of which the best known were the Floating Point Systems products, the AP-120B, the FPS-164 and the FPS-264. The next generations of products were the mini-supercomputers: Multiflow's Trace series of machines and Cydrome's Cydra 5 and the Culler machine for which, as far as we are aware, there is no published description in the literature.

Over the last few years, the VLIW architecture has begun to show up in microprocessors.

Other types of processors with independence architectures have been built or proposed. A superpipelined machine may issue only one operation per cycle, but if there is no superscalar hardware devoted to preserving the correct execution order of operations, the compiler will have to schedule them with full knowledge of dependencies and latencies. From the compiler's point of view,these machines are virtually the same as VLIWs, though the hardware design of such a processor offers some tradeoffs with respect to VLIWs. Another proposed independence archilfCture, dubbed Horizon, encodes an integer H into each operation. The architecture guarantees that all of the next H operations in the instruction stream are data-independent of the current operation. All the hardware has to do to release an operation, then, is assure itself that no more than H subsequent operations are allowed to issue before this operation has completed.

The hardware does all of its own scheduling, unlike VLIWs and deeply pipelined machines which rely on the compiler, but the hardware is relieved of the task of determining data dependence. Below table summarizes the comparison of the instruction-level parallel architecture types:

Table 1

	Sequential Architecture	Dependence Architecture	Independence Architecture
Additional	None	Complete	Minimally, a partial list of

	Sequential Architecture	Dependence Architecture	Independence Architecture
Additional information required in the program	None	Complete specification of dependences between operations	Minimally, a partial list of independences. Typically, a complete specification of when and where each operation is to be executed
Typical kind of ILP processor	Superscalar	Dataflow	VLIW
Analysis of dependences between operations	Performed by hardware	Performed by the compiler	Performed by the compiler
Analysis of independent operations	Performed by hardware	Performed by hardware	Performed by the compiler
Final operation scheduling	Performed by hardware	Performed by hardware	Typically, performed by the compiler
Role of compiler	Rearranges the code to make the analysis and scheduling hardware more successful	Replaces some analysis hardware	Replaces virtually all the analysis and scheduling hardware

Table 1 showing the comparison of the instruction-level parallel architecture types

IJRDET

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

Below table shows the summary of various forms of parallelism:

Table 2

Specification of parallelism	*Execution model*	*Language*	*Parallel architecture*
Loops	Vector execution	Conventional procedural	Vector computers
Loops	SPMD	Conventional procedural	MIMD architectures with barrier synchronization
(1) Explicit declaration of parallel data structures (2) Explicit allocation of parallel data structures to processors	SIMD	Data-parallel	SIMD (array processors)
No explicit specification	Instruction-level function-parallel	Imperative	ILP-architectures
No explicit specification	Data driven	Dataflow	Multi-threaded architectures
(1) Explicit partitioning of program into parallel processes (2) Explicit synchronization among processes	Processes communicating through shared data	Concurrent	Shared memory MIMD machines
(1) Explicit partitioning of program into parallel processes (2) Explicit messages among processes (3) Explicit allocation of processes to processors	Processes communicating by message passing	Parallel	Distributed memory MIMD machines

Table 2 showing the summary of various forms of parallelism

VII. Application Of Parallel Processors

The preparation of programs for parallel execution is of immense practical importance. We have seen that the designs of parallel processing hardware, particularly the interconnection architecture, and of parallel algorithms are full of challenging problems. As difficult as these problems are, the single most important reason for the lack of acceptance and limited application of parallel processing is neither hardware nor algorithms but rather obscure and cumbersome programming models. Since the early 1970s, we have witnessed moderate successes with parallelizing compilers, which automatically extract parallelism from essentially sequential specifications, and with array languages such as High-Performance Fortran. However, the goal of simple, efficient machine-independent parallel programming has remained elusive.

VIII. Conclusion And Future Research

This research paper has developed a framework for integrated code generation with algorithms to optimally solve the main tasks of code generation in a single and fully integrated optimization step for regular and irregular architectures, using dynamic programming and integer linear programming.

It first provided the concept of time profile and the compression theorem for regular architectures. The dynamic programming algorithm for super-scalar and regular VLIW processors is suitable for small and medium-sized problem instances. Secondly, for irregular architectures we introduced the concept of space profiles to describe data locality information and provided the compression theorem for irregular architectures. The dynamic programming algorithm for clustered VLIW processors is applicable to small but not trivial problem instances.

Spilling to memory modules is currently not considered, as we assume that the register classes have enough capacity to hold all values of interest. However, this is no longer true for small residence classes, as e.g. in the case of the Motorola MC56K. In principle our algorithm is able to generate optimal spill code and take this code already into account when determining an op- timal schedule. On the other hand, taking spill code into consideration may considerably increase the space requirements.

The dynamic programming method is suitable for optimizing for time, space, energy, and mixed goals. An energy model from the literature has been adopted and presented a framework for energy-optimal integrated local code generation. A suitable power profile has been defined, which is the key to considerable com- pression of the solution space in our dynamic programming algorithm. The method presented is generic and not limited to a fixed power model. If more influence factors are to be considered that are known at compile time, it can easily be adapted by modifying the power profile definition accordingly.

In order to address larger problem instances we described an optimization technique that exploits partial symmetries in DAGs for regular architectures. The idea of pruning partial solutions that can be shown to be equivalent to others by analyzing local symmetries in scheduling situations did not lead to substantial improvements because the (moderate) size reduction of the solu-tion space was outweighed by the symmetry analysis time. Partial-symmetry optimization techniques still need further investigations and extensions for irregular architectures.

International Journal of Recent Development in Engineering and Technology
Website: www.ijrdet.com (ISSN 2347 - 6435 (Online) Volume 2, Issue 2, February 2014)

Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity of fully integrated code generation. It has been showed that limiting the number of alternatives generated from partial solution nodes (ESnode) produces code with still high quality in much shorter computation times. Further, this thesis provided an integer linear programming formula- tion for fully integrated code generation for VLIW architectures that includes instruction selection, instruction scheduling and register allocation. We have implemented the generation of data for the ILP model within the OPTIMIST framework and compared it to the dynamic programming approach. Currently, the ILP formulation lacks support for memory dependences and for irregular architecture characteristics, such as clustered register files, complex pipelines, etc.

It is intended to extend the formulation as part of future work. Finally, we addressed the issue of retargetable code generation. An architecture description language called xADML has been developed that is based on XML. It successfully retargeted the OPTIMIST framework to three very different processors: ARM9E a single issue processor, TI-C62x a multi-issue clustered VLIW architecture, and Motorola MC56K a multi-issue DSP pro- cessor. As future work, it is needed to address the problem of overlapping register classes and extend our approach beyond local code generation. Some ideas for extending the DP approach to software pipelining have been described. Spilling to memory modules is currently not considered, as we assume that the register classes have enough capacity to hold all values of interest. However, this is no longer true for small residence classes, as e.g. in the case of the Hitachi SH3DSP. The algorithm presented is, in principle, able to generate optimal spill code and take this code already into account for determining an optimal schedule. On the other hand, taking spill code into consideration may considerably increase the space requirements. We plan to develop further methods for the elimination of uninteresting alternatives. Note that our algorithm automatically considers spilling to other register classes already now. Large DAGs require heuristic pruning of the solution space to cope with the combinatorial complexity. This could, for instance, be achieved by limiting the number of ESNodes per cell of the three-dimensional solution space. We did not yet really exploit the option of working with a lattice of residence classes that would result from a more general definition of residence classes based on the versatility relation. This is an issue of future research. In contrast to the limit studies, some people have built real or simulated ILP systems, and have measured their speedup against real or simulated non-parallel systems.

When simulated systems have been involved, they have been relatively realistic systems, or systems that the researchers have argued would abstract the essence of realistic systems in such a way that the system realities should not lower the attained parallelism. Thus these experiments represent something closer to true lower bounds on available parallelism. Ellis used the Bulldog Compiler to generate code for a hypothetical machine. His model was unrealistic in several aspects, most notably the memory system, but realistic implementations should have little difficulty exploiting the parallelism he found. Ellis measured the speedups obtained on 12 small scientific programs for both a "realistic" machine (corresponding to one under design at Yale), and an "ideal" machine, with limitless hardware and single-cycle functional units. He found speedups ranging from no speedup to 7.6 times speedup for the real model, and a range of 2.7 to 48.3 for the ideal model. In this research paper, there are three platforms that add to our understanding of the performance of ILP systems. The paper by Hwu, et al. [lGl}, considers the effect of a realistic compiler which uses superblock scheduling. Lowney, et al. and Schuette and Shen compare the performance of the Multiflow Trace 14/300 with current microprocessors from MIPs and IBM, respectively. Fewer studies have been done to measure the attained performance of software pipelining. Wharter et al. consider a set of 30 doallioops with branches found in the Perfect and SPEC benchmark sets. Relative to a single-issue machine without modulo scheduling, they find a 6 times speedup on a hypothetical 4-issue machine, 10 times speedup on a hypothetical 8-issue machine. Lee, et al., combined superblock scheduling and software pipelining for a machine capable of issuing up to 7 operations per cycle. On a mix of loop-intensive (e.g., LINPACK) and "scalar" (e.g., Spice) codes, they found an average of 1-4 operations issued per cycle, with 2-7 operations in flight.

We hope that this research paper will help in shaping future research of processor systems by providing a more concise view and problem definition, design requirements and constraints, and suggestions for possible research directions.

REFERENCES

[1] C. V. Ramamoorthy and M. J. Gonzalez. A survey of techniques for recognizing parallel processable streams in computer programs. Proc. AFIPS Fall Joint Computin2 Conference (1969).

[2] S. Jain. Circular scheduling: a new technique to perform software pipelining. Proc, ACM SIGPLAN '91 Conference on Pro~rammin2 Lan~ua~e Desi~n and Implementation (June 1991).

[3] 193, M. Smotherman, S. Krishnamurthy, P. S. Aravind and D. Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. Proc. 24th Annual International Workshop on MicrQarchitecure (Albuquerque, New Mexico, November 1991).

[4] S. Ramakrishnan. Software pipelining in PA-RISC compilers. Hewlett-Packard Journal (July 1992).

[5] R. Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions, JQurnal Qfthe ACM 17 (October 1970).

[6] Bengt Johnsson, Bertil Andersson: The Human- Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

[7] H. Jan Komorowski: A Specification of an Ab- stract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

[8] René Reboh: Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91- 7372-489-0. 222

[9] Östen Oskarsson: Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527- 7.

[10] Hans Lunell: Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

[11] Andrzej Lingas: Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

[12] Peter Fritzson: Towards a Distributed Programming Environment based on Incremental Compila- tion,1984, ISBN 91-7372-801-2.

[13] The raw data and more information is available at the follow- ing two URLs:. http://www.pdl.cmu.edu/FailureData/ and http://www.lanl.gov/projects/computerscience/data, 2006.

[14] X. Castillo and D. Siewiorek. Workload, performance, and reliability of digital computing systems. In FTCS-11, 1981.

[15] J. Gray. Why do computers stop and what can be done about it. In Proc. of the 5th Symp. on Reliability in Distributed Software and Database Systems, 1986.

[16] J. Gray. A census of tandem system availability between 1985 and 1990. IEEE Trans. on Reliability, 39(4), 1990.

[17] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In Proc. of ACM SIGMETRICS, 2002.

[18] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. ACM Trans. Comput. Syst., 4(3), 1986.