Exploiting Compiler-Generated Schedules for Energy Savings in High-Performance Processors

Madhavi Valluri
Laboratory for Computer Architecture
The University of Texas at Austin
Austin, TX 78712
valluri@ece.utexas.edu

Lizy John
Laboratory for Computer Architecture
The University of Texas at Austin
Austin, TX 78712
ljohn@ece.utexas.edu

Heather Hanson
Computer Architecture and Technology Laboratory
The University of Texas at Austin
Austin, TX 78712
hhanson@ece.utexas.edu

ABSTRACT

This paper develops a technique that uniquely combines the advantages of static scheduling and dynamic scheduling to reduce the energy consumed in modern superscalar processors with out-of-order issue logic. In this *Hybrid-Scheduling* paradigm, regions of the application containing large amounts of parallelism visible at compile-time completely bypass the dynamic scheduling logic and execute in a low power static mode. Simulation studies using the Wattch framework on several media and scientific benchmarks demonstrate large improvements in overall energy consumption of 43% in kernels and 25% in full applications with only a 2.8% performance degradation on average.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: RISC/CISC, VLIW Architectures

General Terms

Performance, Design

Keywords

Low Energy, Instruction-Level Parallelism, Dynamic Issue Processors, Very Long Instruction Word Architectures

1. INTRODUCTION

A large portion of the energy consumed in modern superscalar processors such as the DEC Alpha 21264, Pentium Pro, Pentium 4, HAL SPARC64, HP PA-8000 etc, can be attributed to the dynamic scheduling hardware (or out-of-order issue logic) responsible for identifying multiple instructions to issue in parallel. Comprising of highly associative and multi-ported queues such as the instruction window and reorder buffer in addition to complex logic circuitry required for the wake-up and select of instructions, the dynamic issue hardware consumes a significant amount of the microprocessor's energy budget. The energy consumption of the dynamic issue hardware accounts for nearly 30% of the overall energy in existing processors [2][4][7], and is projected to grow even further with increasing issue widths and window sizes [20]. A recent study showing the power distribution estimation for the Alpha 21464 microprocessor shows that the issue logic is responsible for nearly 46% of the total power dissipation on the chip [18].

In processors with dynamic scheduling logic, the hardware searches for parallel instructions, irrespective of whether the compiler-generated schedule is perfect or not. The complex, power-hungry dynamic scheduling hardware is perfectly justifiable in many applications with irregular control structures, where it is difficult for the compiler to derive compact schedules due to unpredictable branches and small basic blocks. However, for regular and well-structured programs such as media and scientific applications, the compiler is easily able to generate efficient schedules for considerable portions of the code. The dependence analysis and scheduling performed by the hardware is completely redundant in these regions. Therefore for these regular regions, a large amount of energy is being expended repeatedly in the dynamic issue hardware for unnecessary work.

We present a novel technique that combines the advantages of both compile-time static scheduling and run-time dynamic scheduling to lower the energy consumption in a processor. In this Hybrid-Scheduling paradigm, regions of code containing large amounts of parallelism that can be identified and exploited at compile-time bypass the out-of-order issue logic and are issued and executed exactly in the order prescribed by the compiler. The processor runs in the superscalar mode with dynamic scheduling until a special instruction that indicates the beginning of a statically scheduled (S-Region) is detected, at which point, the out-of-order issue engine is shut down, and the processor switches to a VLIW-like *static mode* in which instruction packets scheduled by the compiler are issued sequentially in consecutive cycles with minimal hardware support. Energy is conserved primarily by reducing the work done in the out-of-order issue logic.

The hybrid-scheduling scheme is particularly suited for high-performance general-purpose systems which need to

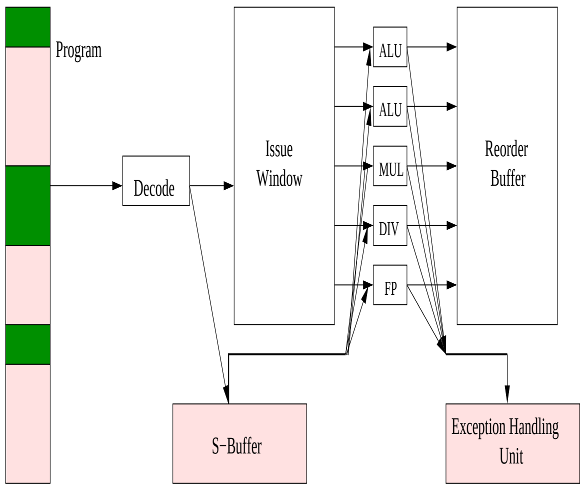

Figure 1: The Hybrid-Scheduling Microarchitecture

cater to diverse application domains such as integer, media and scientific applications. It is important to tune the system to the varying needs of the diverse applications. The hybrid-scheduling architecture thus allows us to use aggressive and power-hungry scheduling hardware for applications that warrant it, while facilitating low energy execution for structured applications that do not need such hardware.

We evaluate the effectiveness of this scheme for several media and scientific benchmarks using the Wattch 1.0 [2] simulator. Our results reveal that we can achieve large improvements in overall energy consumption amounting to 43% in kernels and 25% in full applications with minimal performance degradation.

The rest of the paper is organized as follows: The details of microarchitecture supporting the hybrid-scheduling scheme are described in Section 2. The compiler support required for this scheme is described in Section 3. Our experimental framework and benchmarks used are explained in Section 4. Section 5 presents our simulation results. Related work is discussed in Section 6 and finally, we present concluding remarks in Section 7.

2. THE HYBRID-SCHEDULING MICROARCHITECTURE

The hybrid-scheduling microarchitecture is shown in Figure 1. In this architecture, statically scheduled regions or *S-Regions* execute in a low power static mode. S-Region instructions are scheduled by the compiler into groups or *packets* of independent instructions that can be issued in parallel. The compiler also annotates regions with special "start-static" and "end-static" instructions to indicate the beginning and the end of S-Regions. Once an S-region is detected, instructions in the region can be issued to the function units without any dynamic dependence checks.

Initial Program Execution: In this scheme, the program begins execution in the normal superscalar fashion, i.e. decoded instructions are dispatched to the instruction window where they wait for their operands, ready instructions are issued to the function units and finally instructions retire in-order from the reorder buffer. Execution continues in the superscalar mode until "start-static", an instruction indicating the beginning of an S-Region, is detected.

The S-Buffer: All instructions following the "start-static" instruction, i.e S-Region instructions are decoded and stored in the S-Buffer (shown in Figure 1). The instructions are placed into the buffer before register renaming is performed. The S-Buffer, similar to a decoded instruction cache [8][12], is a circular buffer that stores instructions in the decoded form. For instructions fetched from this buffer we need not fully repeat the decoding step; only register read is performed. A detailed structure is shown in Figure 2. Each S-Buffer line holds one instruction packet. The packet size is fixed and can be determined at design time. The maximum packet size is the number of function units available in the processor. Each S-Buffer line also contains a PC field where the program counter value of the first instruction in the group is stored, and two special bits (S-bit and B-bit) which are set if the line holds the first instruction of an *S-Region* or if it holds a branch instruction respectively. The buffer also maintains a pointer to the next available entry for filling and consecutive instruction packets are placed in consecutive lines of the S-Buffer. Instructions are placed until the "end-static" instruction is detected indicating that all the instructions belonging to the S-Region have been captured in the buffer.

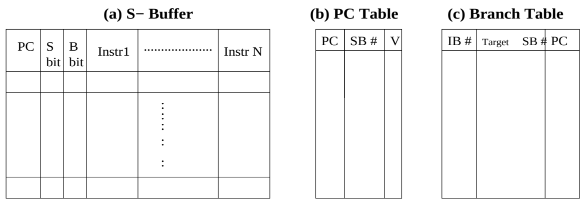

Figure 2: Structures used in the static mode

Switching to Static Mode: After the S-Region has been captured within the S-Buffer, fetch from the instruction cache is stalled and the processor prepares to switch to the static mode of issue. S-Region instructions are scheduled assuming that all the function units are available for use in the static mode and that all the register values are available in the register file, implying that the S-Region instructions can begin execution only after the last instruction in the superscalar mode has completed. Therefore, static mode issue can begin immediately provided the superscalar pipeline has drained while the S-Region was being captured. If not, we wait a few additional cycles for the superscalar pipeline to drain completely. Note that since we have to wait until the superscalar pipeline is empty before we can start issue in the static mode, there is a cycle-time overhead incurred in switching between the two modes. However, if we choose S-Regions such that they execute for long durations in the static mode, the switching cost is amortized, leading to negligible performance degradation. After the last instruction in the superscalar mode has completed, the dynamic issue logic is turned off and instructions begin issuing from the S-Buffer.

Instruction Execution in Static Mode: Instructions from the S-Buffer are issued to the functional units without any further dependence analysis. One complete S-Buffer line is issued every cycle. Memory misses if any, handled by hardware interlocks, cause the static mode issue to stall. All instructions issued in the static mode completely bypass the issue logic, leading to enormous energy savings in the processor. Instructions issue in the static mode until the last instruction of the *S-Region* is executed (detected by the "end-static" instruction). The processor then exits the static mode, starts fetching from the instruction cache and returns to the superscalar mode of execution.

Tracking S-Regions: The scheme uses a small content addressable table, called PC-Table (shown in Figure 2), to keep track of all the S-Region blocks in the S-Buffer. Each entry in the table contains a PC field, the corresponding S-Buffer line number holding the first instruction of the region and a valid bit. Each time a new S-Region is filled into the S-Buffer, an entry is created in the PC table. When a "start-static" instruction is decoded, the PC table is probed to check if the corresponding S-Region is already present in the S-Buffer. Invalidation of S-Regions is handled by adding an extra bit to the instruction buffer, namely the S-bit. The S-Buffer line holding the first instruction in the region has its S-Bit set. The value of the S-bit is always checked before filling a line in the S-Buffer. If the bit is set, we invalidate the entry of the previous S-Region in the PC Table before proceeding any further.

Branching in the Static Mode: Branches in the static mode by default are predicted as 'always taken'. The Branch Table shown in Figure 2 is used for storing the target address of the branch instruction. Entries in the branch table are created when the branch is decoded. Each entry holds the PC of the branch and the S-Buffer line number of the target instruction. The branch table is content addressable with an S-Buffer line number. When issuing an S-Buffer line containing a branch instruction (indicated by the B-bit), the branch table is accessed and the target S-Buffer line number is obtained. Subsequent issue of instructions is performed from this S-Buffer line. If the branch is not-taken, as soon as the branch has been resolved in the execute state, the instruction packet in the decode stage is squashed. The PC Table is searched with the computed target address and if there is no valid entry corresponding to the address, the processor exits the static mode and returns to the superscalar mode of execution. This branching scheme allows us to have zero-cycle branch instructions in the taken case, and a unit cycle latency for the not-taken case when the target instruction is in the S-Buffer. Hence, this scheme is particularly suited for branches highly biased in the 'taken' direction. The hybrid-scheduling compiler employs *if-conversion* [1] to eliminate unbiased branches wherever possible.

In-order Retirement and Exceptions: In the static mode, instructions are issued in packetized groups and also commit as a group. This feature allows us to design a low-power reorder buffer with very few ports and low associativity. Therefore, rather than using the reorder buffer present in the superscalar mode, we provide a separate buffer for the static mode and refer to it as the *Exception Handling Unit* (EHU). We use the reorder buffer with future file approach proposed for VLIW processors by Ozer *et al.* [14] to support in-order commit of instructions and for handling precise exceptions in the static mode.

This section discussed in detail the microarchitecture of the hybrid-scheduling scheme. The following section describes how the compiler selects regions for issue in the low power static mode.

3. ROLE OF THE COMPILER: IDENTIFYING AND SCHEDULING S-REGIONS

S-Regions bypass the dynamic scheduling logic, hence it is critical that the compiler generates schedules comparable to the schedules generated by the out-of-order issue logic for these parts of the code. An S-Region could be a basic block or a group of basic blocks such as loops, hyperblocks [11], superblocks [19] or subroutines. There are several criteria that must be considered before a region can selected for issue in the static mode:

1. The region should exhibit large amounts of ILP (instruction level parallelism), *visible and exploitable at compile-time.* Typical examples of such regions include regions without function calls, code sequences without hard-to-predict branches etc.

2. It is desirable for the region to have a single entry point. This greatly simplifies the task of keeping track of all S-Regions in the S-Buffer. Multiple entry points will require a larger PC-Table to keep track of the regions in the S-Buffer.

3. Regions should have regular, predictable memory access patterns. Due to the absence of dynamic scheduling in the static mode, it is difficult to hide cache miss latencies by instruction overlap. Hence, memory access patterns of the region are critical. Regions with regular access patterns that are amenable to techniques such as prefetching should be chosen.

4. S-Regions of long durations are preferred since there is an overhead for switching from dynamic to static mode as described in the previous section. With long running S-Regions, this cost becomes negligible. The minimum duration of the region is determined by the switching overhead. For example, in our experiments, the switching overhead is observed to be between 20-30 cycles. In order to keep the overhead below 1% the S-Region should run for at least 2000-3000 cycles. Eligible regions can be profiled and regions of sufficient durations can be selected for the static mode of execution.

The compiler schedules instructions in the selected S-Regions into fixed-size packets. To make the schedules compact, techniques such as unrolling, software pipelining, trace scheduling etc are employed. All load and store instructions are scheduled assuming they hit in the cache. Misses, if any, are handled by hardware interlocks. The compiler also annotates the S-Regions with the special "start-static" and "end-static" instructions.

4. EXPERIMENTAL EVALUATION

4.1 Benchmarks

Media and scientific applications are both important domains of applications targeting the high-performance general-purpose processors and these programs have tremendous potential for applying the hybrid-scheduling technique. We study five audio and video compression/encoding applications in the Mediabench [10] suite (*adpcm, epic, g.721, jpeg, mpeg2*) and several media kernels (*iir, add, scale, autocorr,* *fir, dct*). We also study two scientific applications from the SPECFP suite of benchmarks (*swim, tomcatv*).

In our study, we choose loops as static regions and consider all loops without function calls to be potential S-Regions. Loops in the programs were identified and profiled. Table 1 shows the characteristics of the S-Regions in the applications. Column 2 and 3 show the number of loops without function calls in the benchmarks. Column 4 in the table shows the average duration of the potential S-Regions. The duration shown is the weighted average, where the weights are determined based on the percentage of program time attributed to a region. The candidate S-Regions were selected based on the duration of the loop. We set the minimum duration of a loop that could be selected as an S-Region to 1000 cycles, corresponding to approximately 3% switching overhead. Further, loops that exhibited irregular (non-constant stride) memory access patterns were eliminated. We observe that except in G.721, the programs spend a significant amount of time in S-Regions. In G.721, all the S-Regions were disqualified due to their short durations and hence we could not apply the hybrid-scheduling mechanism to this benchmark.

Benchmarks were compiled on a DEC Alpha 21064 machine with the *cc* compiler. The benchmarks are compiled with the highest optimization levels; optimizations such as loop unrolling, software pipelining, if-conversion, prefetching were applied to create compact schedules.

Table 1: S-Regions in Media and Scientific Applications

Benchmark	Number of S-Regions	%Time in S-Regions	Avg. Duration (cycles)	Number of selected S-Regions	Time in selected S-Regions
ADPCM	1	99	17607	1	99
EPIC	18	87	1.2e6	11	87
G721	5	49	84	0	0
MPEG	77	81	6107	32	75
JPEG	20	50	6403	4	31
SWIM	14	93	3.3e8	13	67
TOMC	6	93	2.1e7	4	71

4.2 Simulation Environment

We have implemented the hybrid-scheduling architecture within the Wattch 1.0 simulator [2] framework. Complete configuration details of the simulated processor are given in Table 2. The base processor has an issue width of four. Power distributions for different hardware structures in the base processor are shown in Table 3. The power breakdowns in the table represent the maximum power per unit. We assume that aggressive clock-gating is employed and hence power is scaled linearly with port or unit usage. Unused units dissipate 10% of their maximum power.

The static mode structures are modeled using the RAM and CAM models provided by Wattch 1.0. We assume that the static mode also supports execution of only four instruction per cycle. The structures introduced for the static mode execution are inherently low energy structures due to small sizes, low associativity and fewer port requirements. More details of the structures introduced are given in Table 4. The size of the S-Buffer needs to be large enough to at least hold the largest S-Region in the programs. For the benchmarks studied, an S-Buffer size of 128 lines was found to be sufficient (the largest S-Region in the applications was only 34 lines long, found in *tomcatv*.) Each S-Buffer line contains 4 instruction entries. Each line in the buffer also holds a PC and two state bits. The S-Buffer is accessed only once per cycle, either during filling or during issue.

Table 2: Processor Configuration

Feature	Attributes	Feature	Attributes
IW/LSQ	128/64 entries	S-Buffer	128 rows, 194 bits/row
ROB	128 entries	EHU	16 rows, 284 bits/row
Width	4-way	PC/Br Table	16/10 entries
L1 Dcache	32K 4-way 1-cycle	IALU (4)	1-cycle latency
		FPALU (4)	2-cycle latency
L1 Icache	32K DM 1-cycle	IMult (2)	3-cycle mult lat. 10-cycle div lat.
L2 Cache	512KB, 4W 8-cycle	FPMult (2)	3-cycle mult lat. 15-cycle div lat.
Memory	40-cycle lat	LD/ST (2)	1-cycle

Table 3: Power distribution for different hardware structures in the baseline processor. Total processor power for this configuration is 125W.

Unit	Power	Unit	Power
Branch Predictor	3.49%	Rename Logic	0.46%
IW/LSQ	15.45%	ROB	19.70%
Register File	2.63%	Result Bus	3.98%
Int ALU	3.42%	FP ALU	10.5%
ICache	2.61%	ITLB	0.20%
DCache	5.22%	DTLB	0.68%
L2 Cache	3.42%	Clock	28.20%

The size of the exception handling unit is also small, since it needs to be only one entry longer than the longest latency operation [14]. Instructions access the EHU associatively during instruction writeback. However, unlike the reorder buffer, during issue and commit only a single entry is accessed [14]. Each row in the EHU contains four instruction entries. Each entry holds one result value, a destination register number and two state bits. The maximum power consumed by the EHU is 1.1% and the future file, which is similar to the register file, is 2.63% of the overall processor power. The PC and branch tables are small structures since we place only few S-Regions in the S-Buffer. They account for only 0.24% of the total processor power.

Table 4: Static Mode Structures

Unit	Entries	bits/row	Ports	Max access	Assoc. access	Power(% of total)
S-Buffer	128	194	1R/1W	1/cyc	No	0.68%
EHU	16	284	1R/5W	6/cyc	Partial	1.1%
Future File	32	64	8R/4W	12/cyc	No	2.63%

5. EXPERIMENTAL RESULTS

Figure 3 provides the energy and energy-delay results for all benchmarks. The corresponding performance degradation suffered by the programs is given in Figure 4. We observe that the hybrid-scheduling scheme is able to achieve very large improvements in energy consumption without any significant increase in the execution time.

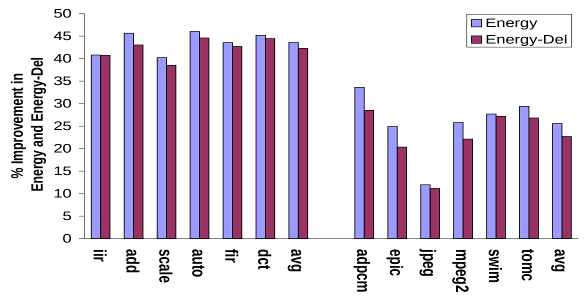

Figure 3: Energy and Energy-Delay improvements

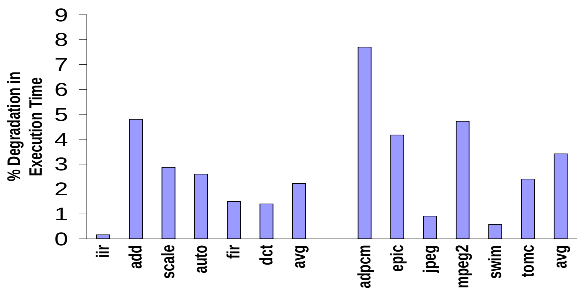

Figure 4: Performance degradation results

In kernels, the average energy improvement is seen to be 43.6%, with the improvement ranging from 40% (`scale`) to 46% (`autocorr`). The average performance degradation caused by the hybrid-scheduling approach is a mere 2.23%. The highest performance drop of 4.8% is observed in `add` and the lowest is 0.16% seen in `iir`. On an average, the energy-delay product improves by 42.3%.

The energy improvement and performance results for the applications are also included in Figures 3 and 4. On average, in applications, we observe an energy improvement of 25% and a performance degradation of 3.4%. In the applications, the energy savings are directly proportional to the amount of time spent in S-Regions. The highest improvement in energy is seen in ADPCM (33%), in which almost 99% of the execution time is spent in S-Regions. JPEG shows the lowest savings (12%).

The energy improvements seen are primarily due to the savings in the issue window and reorder buffer power. Additional power savings are observed in the fetch and decode phases of the pipeline. Since static mode instructions are accessed from the S-Buffer which is significantly smaller than the instruction cache, fetch power reduces considerably. Further, since instructions are stored in a decoded form, decoder power is also saved in the static mode. Additionally, we do not access the branch predictor in the static mode, this leads to further energy savings. Figure 5 shows the energy savings in each hardware structure. Energy savings in the clock nodes of the structures is shown separately. Note that these are not absolute values but only portray the ratio of savings from each structure.

The performance degradation suffered by an application depends on the nature of the loop schedules, static mode cache misses and the switching overhead incurred. One of the key constraining factors observed in this study is the limitation that the schedules generated by the compiler are restricted to integral values. The performance degradation caused due to rounding the lengths of schedules to integer values can be significant but is considerably reduced by un-

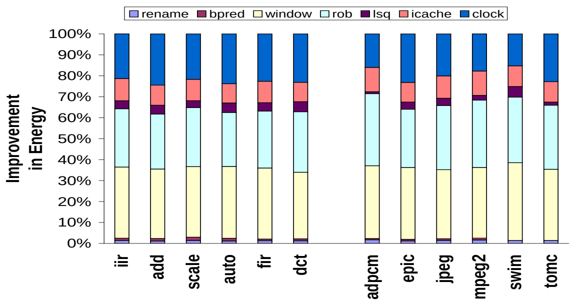

Figure 5: Energy improvements in different hardware structures

rolling the loop. In our experiments we explored different degrees of unrolling and software pipelining to find perfect or near-perfect schedules for the loops. In addition to the integral length schedule limitation, degradation observed in Figure 4 is caused due to the switching overhead and cache misses if any.

6. RELATED WORK

Recent work in the area of energy-effective microprocessors has shown that it is possible to reduce power in the out-of-order issue units and other units by dynamically resizing the structures [3][4][6][15][16]. Some approaches monitor the changes in the IPC (instructions per cycle) of the program and alter the issue window size [3][4]. Ghaisi *et al.* [6] propose a technique wherein the OS dictates the expected IPC of the program for different phases, and the hardware chooses between different processor configurations such as pipeline-gating, in-order issue and out-of-order issue. Another approach uses the immediate history of the actual usages of the different queues to resize them [15]. Dynamic critical path information used for steering critical and non-critical instructions separately to two smaller queues rather than one large queue has shown considerable gains in energy consumption [16]. Iyer et al. explore a technique which profiles different characteristics of the program such as ALU usage, register file usage, instruction window usage, etc [9]. Hotspots in the program are determined and processor units are scaled accordingly.

In all the above techniques, the sizes of issue logic structures are manipulated. In our approach we completely *eliminate* the use of an instruction window and a complex reorder buffer for some regions of the code. Further, all dynamic resizing methods that reduce power in the superscalar pipeline structures could potentially be applied in the superscalar mode of execution in the hybrid-scheduling approach leading to larger overall savings in energy consumption.

Talpes *et al.* [17] suggest a technique that collects schedules created by the dynamic issue logic into a large trace cache and reuses them to save issue power. Franklin *et al.* [5] and Nair *et al.* [13] proposed similar schemes that were aimed primarily at improving the clock frequency of the processor. An important difference between these techniques and the hybrid-scheduling scheme is that these techniques are insensitive to the available ILP in different phases of programs, leading to inefficient use of the caches holding the scheduled instructions. Moreover, schedules created by the dynamic issue hardware, when looking at a limited window of instructions, are not known to be optimal.

7. CONCLUDING REMARKS

In this work, we introduce a hybrid-scheduling technique that conserves energy in a superscalar processor by exploiting compiler-generated schedules for regular and structured regions of applications. Regular regions bypass the power-hungry units such as the issue window and reorder buffer and execute in a low power static mode. Execution in the static mode also results in additional energy savings in the decode logic, instruction cache and branch predictor. The hybrid-scheduling architecture employs dynamic scheduling for the less regular instruction sequences. We show that the hybrid-scheduling technique can reduce energy consumption by as much as 46% for kernels and up to 33% in full-length applications with minimal performance degradation. Further, an attractive feature of this architecture is that it allows us to orthogonally apply all previously proposed dynamic resizing techniques to reduce energy consumption in the superscalar mode of issue for much larger overall energy savings in the processor. The proposed technique will be effective for any application which contains regions of code that can be statically scheduled effectively. While it might be difficult to identify such regions in many general purpose integer applications, the necessity to support media and other applications with structured code on the desktop computer makes this technique important for general purpose microprocessors.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments. We thank Steve Keckler and Doug Burger for many useful discussions during the early stages of this work. We also thank Juan Rubio, Anand Ramachandran and Vivekananda Vedula for their suggestions to improve the initial draft of the paper.

This research is partially supported by the National Science Foundation under grant number 0113105; the Defense Advanced Research Projects Agency under contract F33615-01-C-1892; and by the AMD, Intel, IBM, Tivoli and Microsoft corporations.

9. REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *POPL83*, Austin, Jan 1983.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th International Symposium on Computer Architecture*, Jun 2000.

[3] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computers Systems, held in conjunction with ASPLOS*, Nov 2000.

[4] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *28th International Symposium on Computer Architecture*, Jun. 2001.

[5] M. Franklin and M. Smotherman. A fill-unit approach to multiple instruction issue. In *27th Annual International Symposium on Microarchitecture*, 1994.

[6] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity Effective Design, Vancouver, Canada*, Jun. 2000.

[7] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Design Automation Conference*, pages 726–731, 1998.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. Technical report, Intel, Feb. 2001.

[9] A. Iyer and D. Marculescu. Run–time scaling of microarchitecture resources in a processor for energy savings. In *Kool Chips Workshop, held in conjunction with MICRO–33*, 2000.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *30th International Symposium on Microarchitecture*, pages 330–335, Dec. 1997.

[11] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th International Symposium on Microarchitecture*, pages 45–54, 1992.

[12] S. W. Melvin, M. Shebanow, and Y. N. Patt. Hardware support for large atomic units in dynamically scheduled machines. In *21st International Symposium on Microarchitecture*, Dec. 1988.

[13] R. Nair and M. E. Hopkins. Exploiting instruction level parallelism in processor by caching scheduled groups. In *ISCA*, 1997.

[14] E. Ozer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings, and T. M. Conte. A fast interrupt handling scheme for VLIW processors. In *International Conference on Parallel Architectures and Compilation Technique*, Oct. 1998.

[15] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *34th International Symposium on Microarchitecture*, pages 90–101, Dec 2001.

[16] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[17] E. Taples and D. Marculescu. Power reduction through work reuse. In *International Symposium on Low Power Electronics and Design*, 2001.

[18] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look to the future. In *CoolChips Tutorial, An Industrial Perspective on Low Power Processor Design in conjunction with Micro-33*, Dec. 1999.

[19] W.M.W. Hwu et al.,. The superblock: An effective technique for vliw and superscalar compilation. In *The Journal of Supercomputing 7(1)*, Jan 1993.

[20] V. V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures. In *IEEE Transactions on Computers*, pages 268–285, Mar. 2001.