US 20050114850A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2005/0114850 A1

Chheda et al. (43) Pub. Date: May 26, 2005

(54) ENERGY-FOCUSED RE-COMPILATION OF EXECUTABLES AND HARDWARE MECHANISMS BASED ON COMPILER-ARCHITECTURE INTERACTION AND COMPILER-INSERTED CONTROL

(76) Inventors: **Saurabh Chheda**, Amherst, MA (US); **Kristopher Carver**, Amherst, MA (US); **Raksit Ashok**, Amherst, MA (US)

Correspondence Address:
Sylvia Moritz
BlueRISC, Inc.
28 Dana Street
Amherst, MA 01002 (US)

(21) Appl. No.: **10/967,989**

(22) Filed: **Oct. 18, 2004**

Related U.S. Application Data

(60) Provisional application No. 60/515,260, filed on Oct. 29, 2003.

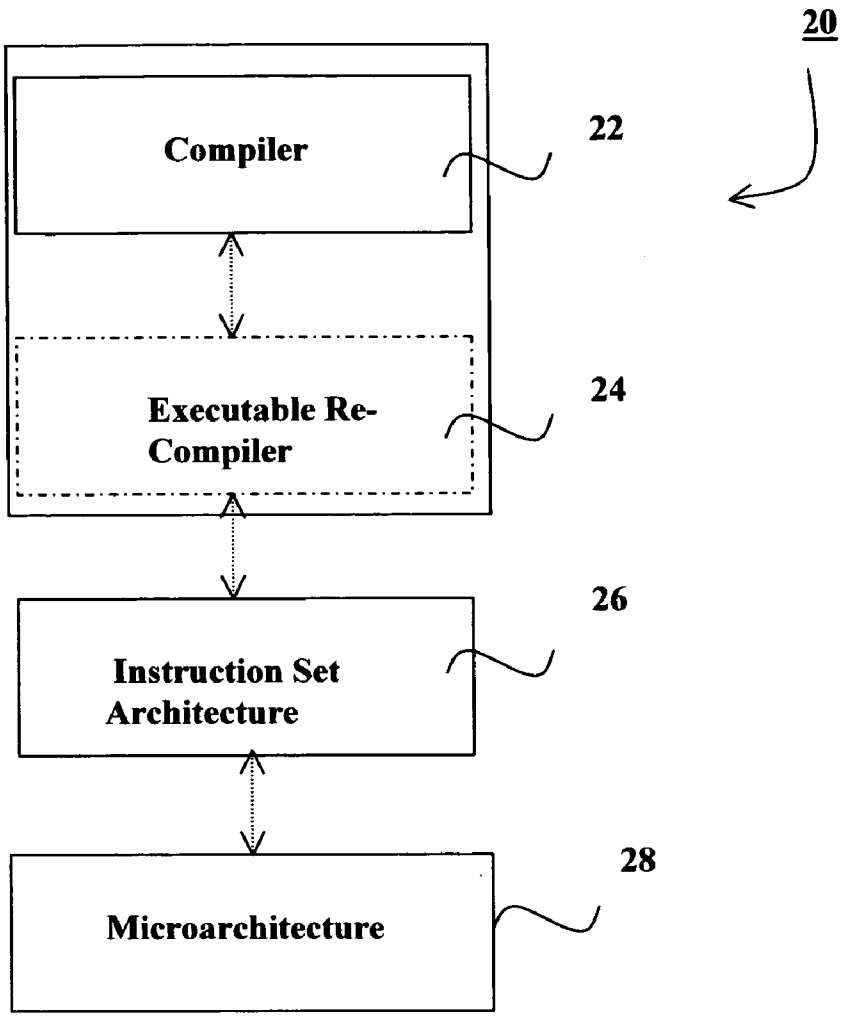
Publication Classification

(51) **Int. Cl.**[7] **G06F 9/45**

(52) **U.S. Cl.** **717/151**

(57) **ABSTRACT**

A method comprising of analyzing and transforming a program executable at compile-time such that a processor design objective is optimized. A method including analyzing an executable to estimate energy consumption of an application component in a processor. A method including transforming an executable to reduce energy consumption in a processor. A processor framework controlled by compiler inserted control that statically exposes parallelism in an instruction sequence. A processor framework to reduce energy consumption in an instruction memory system with compiler inserted control.



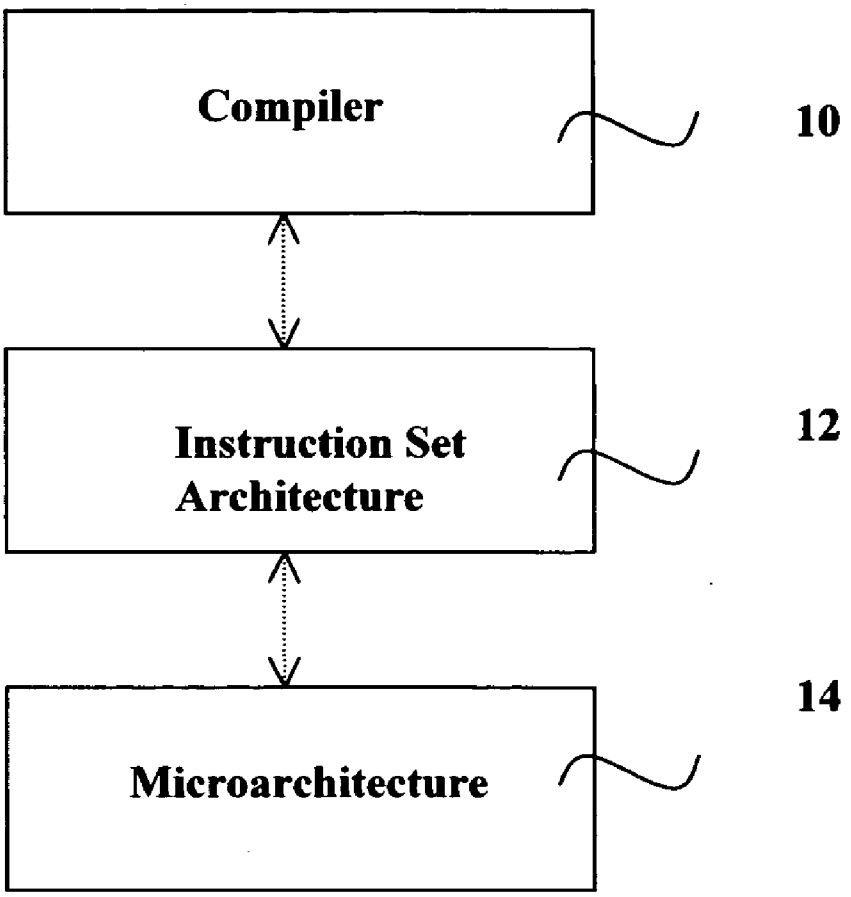

Compiler
10
Instruction Set Architecture
12
Microarchitecture
14

FIG. 1

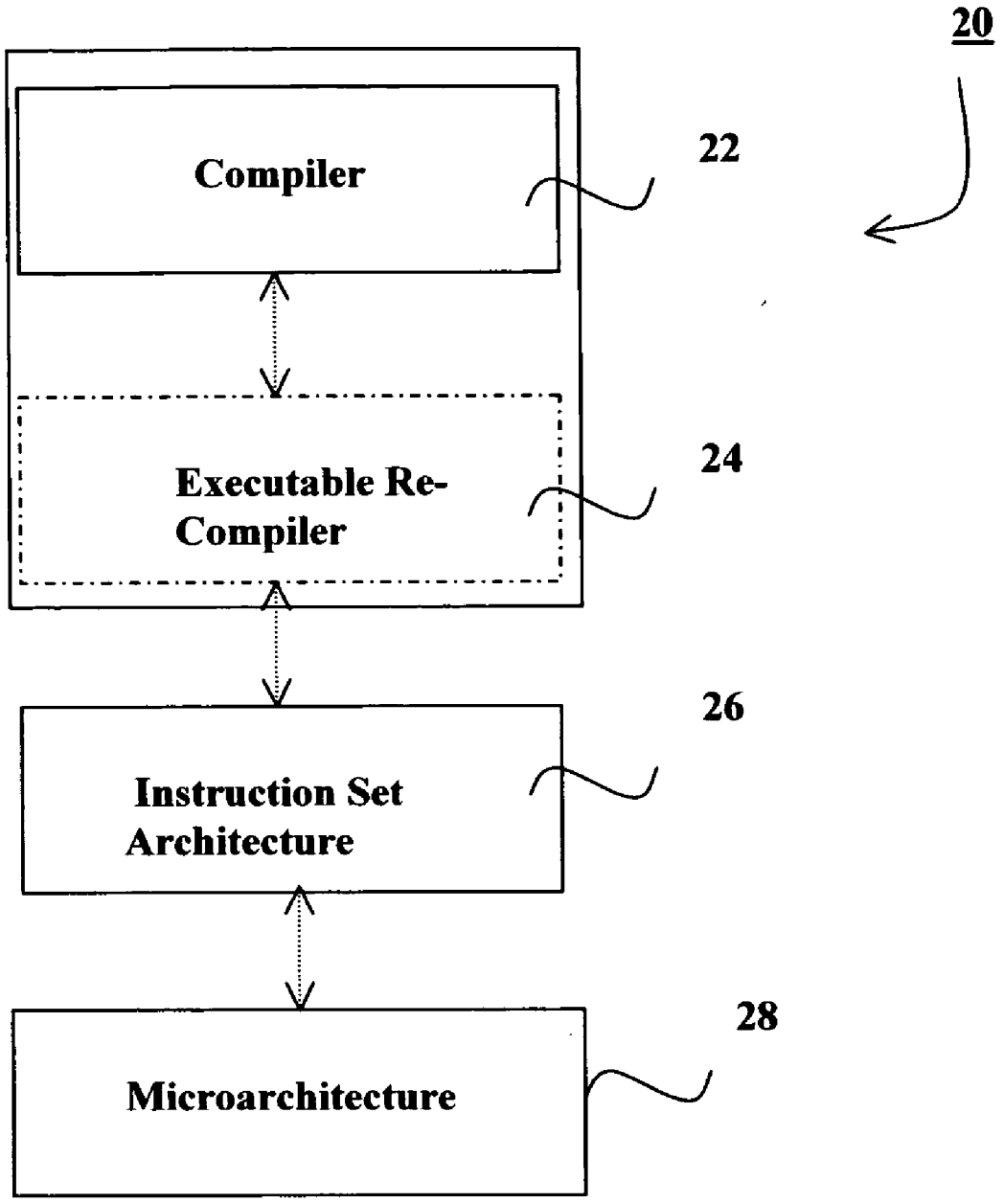

20
Compiler
22
Executable Re-Compiler
24
Instruction Set Architecture
26
Microarchitecture
28

FIG. 2

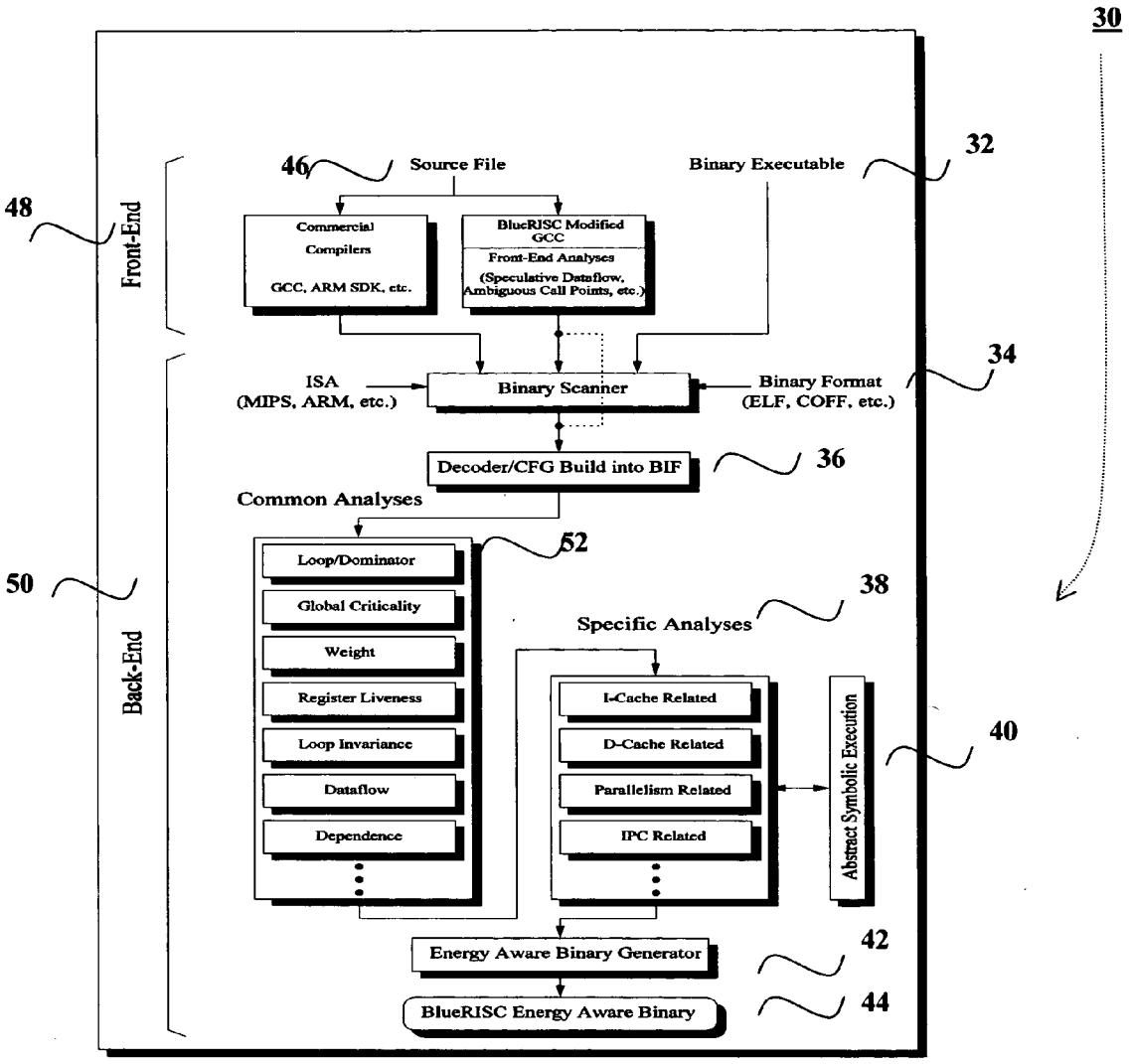

30
32
46
Source File
Binary Executable
48
Front-End
Commercial Compilers
GCC, ARM SDK, etc.
BlueRISC Modified GCC
Front-End Analyses
(Speculative Dataflow, Ambiguous Call Points, etc.)
34
ISA
(MIPS, ARM, etc.)
Binary Scanner
Binary Format
(ELF, COFF, etc.)
Decoder/CFG Build into BIF
36
Common Analyses
52
50
Back-End
Loop/Dominator
Global Criticality
Weight
Register Liveness
Loop Invariance
Dataflow
Dependence
38
Specific Analyses
I-Cache Related
D-Cache Related
Parallelism Related
IPC Related
Abstract Symbolic Execution
40
Energy Aware Binary Generator
42
BlueRISC Energy Aware Binary
44

FIG. 3

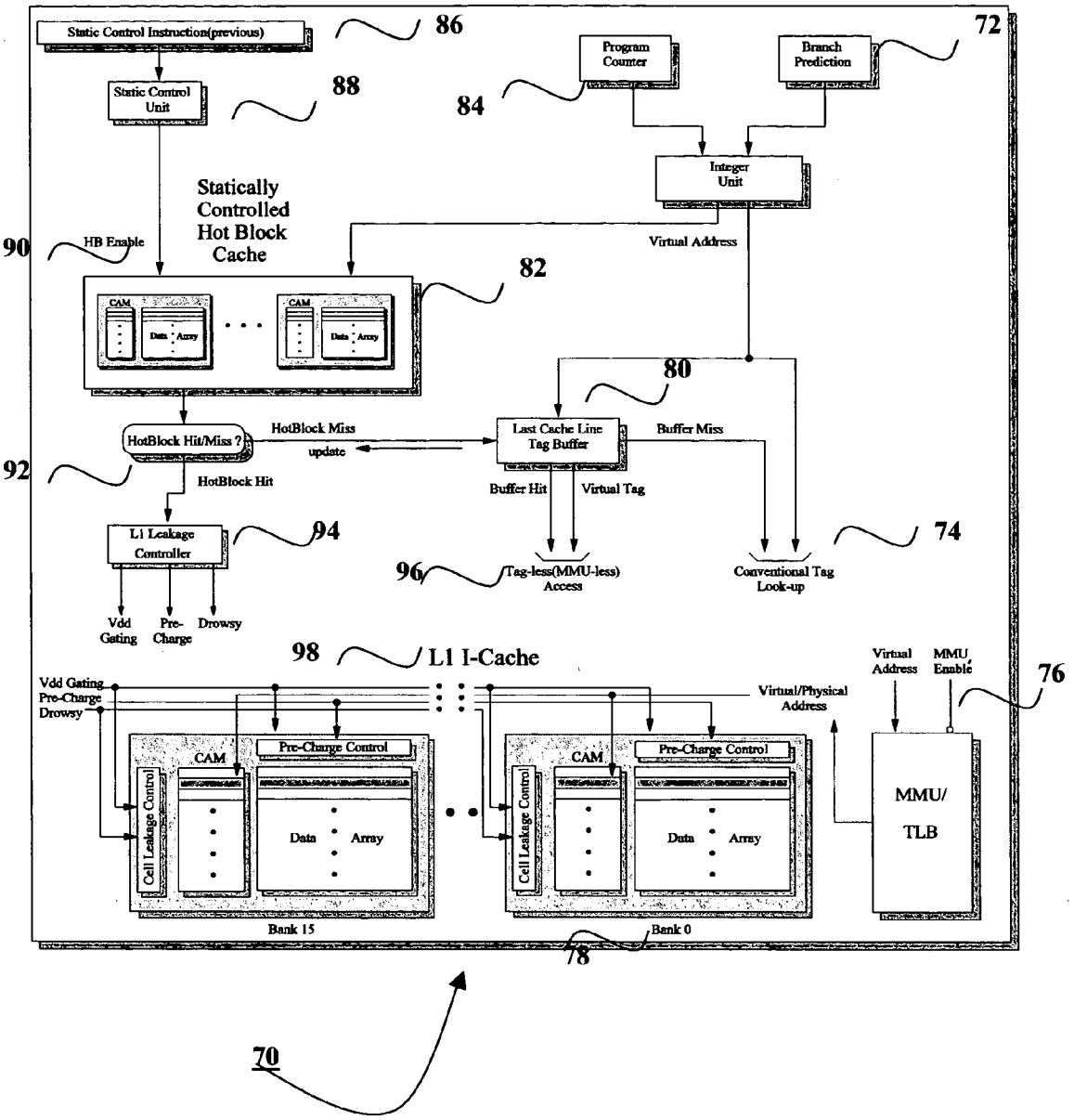

Static Control Instruction(previous)
86
72
Program Counter
Branch Prediction
Static Control Unit
88
84
Integer Unit
Statically Controlled Hot Block Cache
90
HB Enable
Virtual Address
82
CAM
Data Array
80
HotBlock Hit/Miss ?
HotBlock Miss
update
Last Cache Line Tag Buffer
Buffer Miss
92
HotBlock Hit
Buffer Hit
Virtual Tag
L1 Leakage Controller
94
74
96
Tag-less(MMU-less) Access
Conventional Tag Look-up
Vdd Gating
Pre-Charge
Drowsy
98
L1 I-Cache
Virtual Address
MMU Enable
76
Vdd Gating
Pre-Charge
Drowsy
Virtual/Physical Address
Pre-Charge Control
Cell Leakage Control
Data Array
MMU/ TLB
Bank 15
Bank 0
78
70

FIG. 4

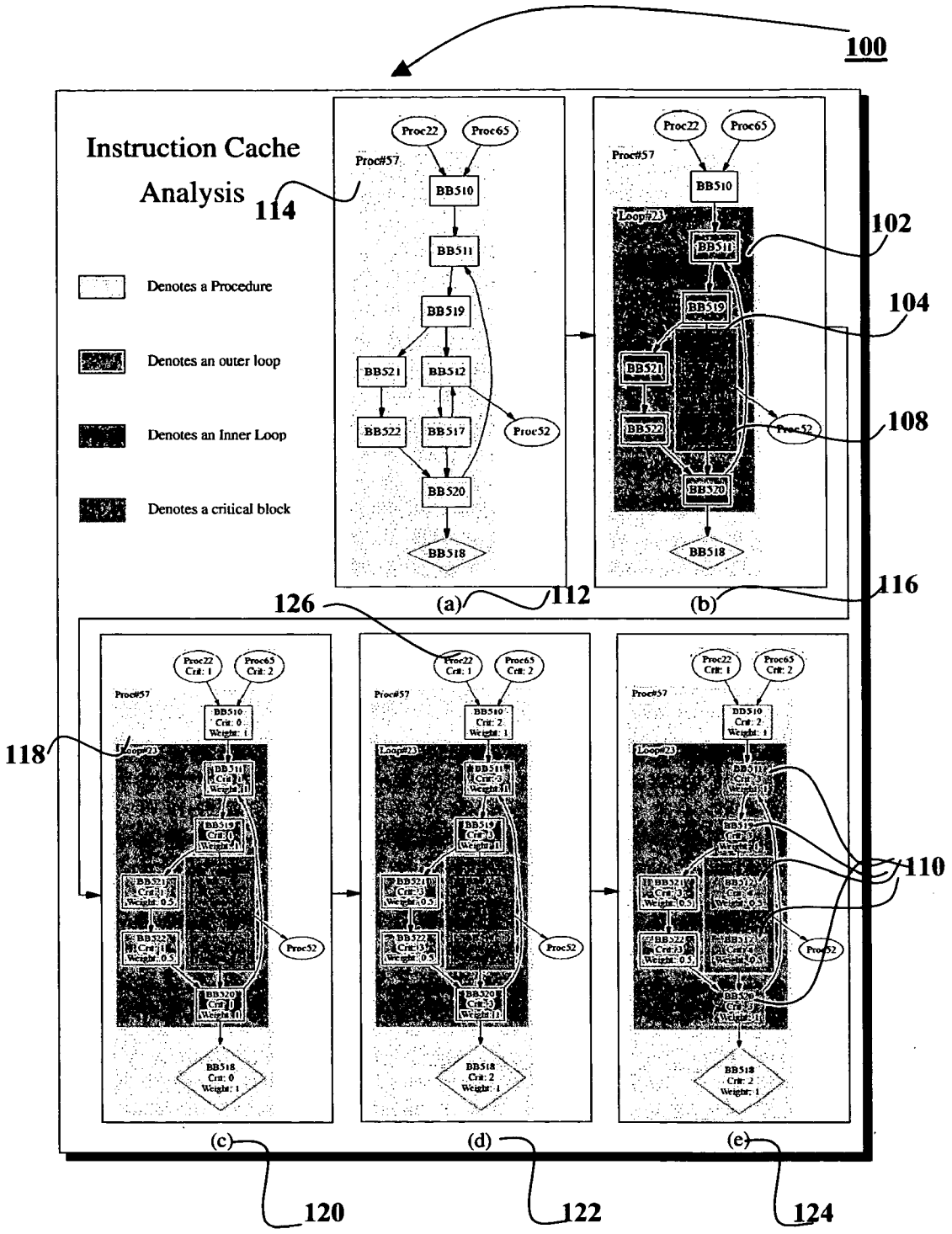

100
Instruction Cache Analysis
114
Denotes a Procedure
Denotes an outer loop
Denotes an Inner Loop
Denotes a critical block
Proc22
Proc65
Proc#57
BB510
BB511
BB519
BB521
BB512
BB522
BB517
Proc52
BB520
BB518
Loop#23
102
104
108
(a)
112
(b)
116
126
118
(c)
120
(d)
122
(e)
124
110

FIG. 5

ENERGY-FOCUSED RE-COMPILATION OF EXECUTABLES AND HARDWARE MECHANISMS BASED ON COMPILER-ARCHITECTURE INTERACTION AND COMPILER-INSERTED CONTROL

RELATED U.S. APPLICATION DATA

[0001] This application claims the benefits of U.S. Provisional Application No. 60/515,260, filed on Oct. 29, 2003, and Confirmation No 8574, entitled: ENERGY-FOCUSED RE-COMPILATION OF EXECUTABLES AND HARDWARE MECHANISMS BASED ON COMPILER-ARCHITECTURE INTERACTION AND COMPILER-INSERTED CONTROL, the contents of which are hereby incorporated by reference into this application as if set forth herein in full.

TECHNICAL FIELD

[0002] This invention relates generally to reducing energy and power consumption and improving performance in a microprocessor and, more particularly, to reducing energy and power consumption and improving performance by modifying a program executable. In addition, it relates to hardware mechanisms that use compiler control inserted into selected sequences of instructions to reduce energy consumption. Furthermore, it relates to compiler-exposed parallelism on compiler-selected program sequences that enables energy-efficient parallel execution with minimal hardware support required.

BACKGROUND

[0003] Microprocessors (referred to herein simply as "processors") consume energy/power during their operation. It is advantageous to reduce the amount of energy consumed, particularly in the case of devices that run off limited power supplies.

[0004] Various factors affect the amount of energy that a processor consumes. For example, the frequency at which the processor operates, the voltage level that powers the processor, as well as the load capacitances affect processor energy consumption. Reducing the frequency of the processor or the voltage supply may decrease processor energy consumption, however, doing so may also adversely affect the performance of the processor.

[0005] Other techniques to reduce energy consumption by, for example, reducing load capacitances, may include changes to processor architectures and processor circuits. Some other techniques rely on modifying the application itself, or any other system layer, to improve energy efficiency.

[0006] An executable is a version of a software application that has been compiled from a programming language into a processor instruction set.

[0007] A source-level compiler transforms source codes into a sequence of instructions based on a processor instruction set.

[0008] Incorporating energy awareness into a source-level compiler is a very complex process; it could also negatively affect performance or some other design objective. This is due to the interactions between optimizations that target different objectives.

[0009] Additionally, not all source codes might be available for source-level compilation of an application, and therefore not all codes could be optimized in order to improve energy efficiency or reduce power consumption.

[0010] Moreover, the modifications would need to be incorporated in all compilers that aim to optimize energy consumption.

[0011] For a given processor there are typically many compilers available provided by many different vendors. These compilers have their own advantages and disadvantages.

[0012] This makes incorporating energy optimizations to source-level compilers a challenging task.

[0013] Accordingly, if energy-awareness is introduced at the executable-level instead, by transforming the executable itself, significant practical advantages could be achieved.

[0014] The goal would be to optimize executables that may have been fully optimized previously with a source-level compiler targeting a design aspect such as performance.

[0015] In general, such an executable-level re-compiling based approach to energy optimizations, could enable keeping optimizations performed during source-level compilation largely intact, could provide access to and optimize all program codes including static and dynamic libraries, and could be used on existing executable codes that have been generated with a variety of different compilers and potentially from different vendors.

[0016] The executable file itself provides a convenient interface between, for example, performance-oriented optimizations and energy-oriented optimizations. Moreover, one energy optimization tool or layer could be used with many different source-level compilers; this does away the need to retrofit all source compilers to optimize energy consumption.

[0017] Another aspect of this invention relates to scalability. Reductions in energy consumption should also be scalable, meaning that they are implemented such that processors having different architectures and instruction sets can easily be targeted. An executable-level re-compiling approach could provide such scalability.

[0018] This aspect may include an energy-aware program representation that encapsulates information reconstructed from executables in an abstract and retargetable manner to achieve scalability.

[0019] Another aspect of this invention relates to how application parallelism can be exploited in processors without significantly increasing load capacitances. If parallelism is achieved but with an increase in load capacitances, due to hardware complexities, the advantage of improved performance is offset by the resultant much higher power consumption. Current state-of-the-art solutions to expose parallelism are not energy efficient. As such, many of today's low-power processors are single issue.

[0020] Incorporating compiler information to enable issuing multiple instructions in parallel with a Very Large Instruction Word (VLIW) format has been shown to be detrimental to energy consumption. The term VLIW refers to the size of each instruction that is executed by a processor.

This instruction is very long in comparison to the instruction word size utilized by most current processors.

[0021] Energy inefficiency in a VLIW processor is often attributed to the fixed wide-issue instruction set format; in many applications or program sequences there is not enough parallelism to fill all the instruction slots available in a VLIW instruction.

[0022] In fact, on average, there is typically very little instruction-level parallelism (ILP) available. As noted in the literature, typical applications have an average ILP level of less than two; thus, a 4-way VLIW would have on average two of its instruction slots unutilized. The unused slots would contribute to unnecessary instruction fetches and instruction-memory energy consumption. Higher ILP levels are fundamentally limited by true data dependencies. While speculation-based techniques can improve ILP levels, runtime speculation has an energy cost that typically offsets the benefits of the higher ILP.

[0023] As noted in the literature, energy consumed by the instruction memory as well as fetch energy are a significant fraction of a processor energy consumption. For example, a state-of-the-art ARM10 processor has been reported to consume 27% of its total energy in the instruction memory system.

[0024] Other systems such as superscalar processors attempt to discover parallelism at runtime with significant hardware support. This support reduces the energy benefits obtained with parallel execution. Simply, the performance benefits are more than offset by the increase in load capacitances that increase power consumption.

[0025] In one aspect, the present invention solves the problem of exposing parallelism without requiring significant hardware support, such as is required in superscalar designs, and without having a fixed wide-issue instruction format, such as in VLIW designs. The solution is adaptive and compiler driven.

[0026] It works by incorporating control bits into the binary to issue instructions in parallel on only selected sequences of instructions, on compiler demand. The approach could be limited to sequences where there is enough parallelism and when is considered or estimated to be good for energy efficiency. Thus, parallelism encoding can be limited to critical program paths, typically a relatively small fraction of the instructions in a binary, to improve energy efficiency.

[0027] In addition, if other compiler-managed optimizations are incorporated, such as for energy reduction purposes in the memory system, the added instruction bits for the various optimizations could be encapsulated into one or more new control instructions or control data. If the control is implemented with instructions, both regular instructions that are extensions to the regular ISA or co-processor instructions can be used.

[0028] In one embodiment, a solution to incorporate control information is to use the co-processor interface, that is, without requiring changes to the processor's regular instruction set. The inserted instructions may be folded, i.e., extracted early, in the prefetch queue before entering the processor pipeline, in a somewhat similar manner to zero-cycle branches in some architectures, e.g., ARM10. Such a solution removes pipeline bubbles that might otherwise be caused by the control instructions. An advantage, therefore, of using co-processor instructions is that one could easily add such control to existing processor cores. In one embodiment that is implemented within an ARM 10 design, each such control instruction would enable the encoding of 21 bits worth of control information.

[0029] Control information may be added per a sequence of instructions, such that the code dilution overhead of static control could be amortized across several different optimizations and for several instructions in the sequence. The sequence where optimization is applied can be determined with static analysis or other means such as profiling. A control instruction is typically inserted before the controlled instruction sequence.

[0030] Energy increase due to the extra fetches can be minimized with compiler-driven instruction memory optimizations, for example, by almost always fetching control bits from more energy efficient smaller cache partitions, driven by compiler decisions. One aspect of this invention demonstrates such capability.

[0031] Due to the compiler-driven nature of the solution, the impact of control overhead can be kept very small. In our experience, in one embodiment, such control energy overhead could be kept below 1%-2% if instruction memory energy optimizations are included, while providing energy optimizations in the range of 30%-68% if several techniques in different processor domains/components are included.

SUMMARY

[0032] The executable-level modification or executable re-compilation, and compiler-architecture interaction based processor framework described herein address the foregoing need to reduce energy consumption in a practical and flexible manner. The approach provides energy savings without adverse effects on performance and with scalability, to different processor instructions sets, and easy integration with other compilation tools. In fact, the framework in its preferred embodiment, when incorporating support for compiler-driven parallel issuing of instructions, improves performance.

[0033] The framework, in one of its embodiments, eliminates the need for reengineering of source-level compilers to target energy efficiency. The framework can be used in, but is not limited to, converting existing executables where source codes are not always available to achieve a more energy efficient operation.

[0034] The executable-level re-compilation approach could also transform static libraries that may be part of an executable, or even dynamic libraries. Sources of such libraries are typically not available at the time of source-level compilation of applications, and, therefore, would not be affected by source-level compilation based energy optimizations. In one aspect, this invention enables the global optimization of all application codes together, including libraries, for example, to target energy reduction.

[0035] In general, there might be several compilers available for one particular processor or an instruction set architecture. The approach based on executables can be easily integrated with some or all of these compilers without requiring changes in the source-level compiler.

[0036] In one aspect, the framework does this by extracting and utilizing static compile-time information from fully linked and optimized executables, and by modifying the executables according to new design goals such as energy efficiency.

[0037] In principle, a source-level compiler could augment an executable to support the energy optimization process based on executable re-compilation. The framework may include support for using such information.

[0038] The framework is also applicable at pre-linking level and on other versions of an executable.

[0039] The framework could be applied on executables that might contain symbolic information that was inserted for debugging purposes, or, for example, targeting energy optimizations.

[0040] The framework could be utilized to perform optimizations that are not limited to energy reduction. Examples of optimizations performed may include, but are not limited to, energy, power, performance and security. In general, in one aspect, the framework could be used to improve an application's performance or improve the application's energy-delay product or any other metric that is a combination of execution time and power consumption.

[0041] The executable-level modifications can be used in combination with added architectural support, or with unchanged architecture, to gain benefits.

[0042] When used without added architectural support the framework optimizes the instruction stream so as to reduce energy consumption or improve performance on an existing hardware solution.

[0043] When used with added architecture support, the framework could provide unique opportunities for compiler-architecture based optimizations, by incorporating support for statically-managed architectural mechanisms in a processor, that are typically much more energy-efficient than conventional dynamic mechanisms.

[0044] Statically-managed access mechanisms use static information to reduce redundancy typically present in conventional instruction execution. In fact, each instruction can be represented as a sequence of micro-operations during execution.

[0045] Example of such micro-operations may include tag checks in caches, multi-way lookups in associative data arrays, virtual memory related address translations, Translation Lookaside Buffer accesses, and register file accesses. Many of these micro-operations are not necessary if there is related information extracted in the compiler that provides it, or if there is program information that enables a way to replace these micro-operations with more energy-efficient but equivalent ones.

[0046] Disambiguating an executable is the process of extracting symbolic information and disassembling program instructions from an executable. In one aspect, the invention is directed to a method, for use in a processor, which includes disambiguating the application executable into an intermediate format, performing a number of analyses on this format to statically determine program characteristics, and changing a program's structure and instructions such that the estimated energy consumption of the program after the changes is less than the energy consumption of the original unmodified program. This aspect may include one or more of the following features.

[0047] The method may include identifying aspects of the program including branches, procedure call points, memory related operations and their locality and reuse patterns, criticality of each instruction in the instruction stream from the point of view of performance, power, and energy consumption, and other similar static information.

[0048] The method may include static executable disambiguation and late code modification techniques and may include using abstract execution, in addition to static analysis, to help the disambiguation process. Abstract execution is a process based on which short sequences of instructions are executed abstractly using an abstract machine.

[0049] Abstract execution may be integrated in the compiler and may operate in a forward and/or backward manner. Abstract execution is a process of running a simulation engine during program analysis, to help disambiguate, for example, program-flow or memory access patterns, but is not limited to only those aspects. While in its preferred embodiment this simulation engine is a very small functional simulator kernel optimized to run in a compiler, the framework is not limited to only such a simulator.

[0050] Furthermore, the method may include inserting a more energy efficient instruction replacing an existing instruction and may include inserting control bits to control hardware structures.

[0051] In one aspect, the control bits and/or instructions may be implemented as extensions to the regular instruction set of a processor, and/or as co-processor instructions, and/or as data.

[0052] The control bits may also be implemented as data added to the executable, data that is associated with-segments of instructions and that are accessed during execution by a processor. This aspect may include dedicated architectural support to map instruction sequences to corresponding control bits efficiently. This may include changes to an architecture structure such as the branch target address cache, that would associate the target address with a target static control data, controlling a sequence of instructions such as a basic block, or super block consisting of several basic blocks, starting after the target address. This data could be part of the instruction memory similar, for example, to how constants are used in many instruction sets. A basic block is a sequence of instructions that has no branch instructions except possibly the last instruction and has only one entry point, from other parts of the code, that is at the top of the block.

[0053] The control bits could control one or several instructions in the instruction stream during execution. A possible objective is to enable energy reduction of these controlled instructions at runtime, by letting the processor know ahead of execution their preferred execution modes to reduce energy.

[0054] If the control bits refer to controlling several instructions during runtime, such as in a basic block, one control instruction inserted before the block may include control information for a variety of optimizations amortizing its added overhead across many optimizations. The perfor-

mance overhead of control instructions could be avoided with early folding/removal in processor pipelines with simple pre-decoding logic added to prefetch buffers and instruction fetching. In many predicated architectures a similar early folding is used for example to achieve zero-cycle branches.

[0055] Another aspect may include optimizing control-bits related fetch energy by compile-time mapping and runtime fetching of the control-bits from an energy optimized memory structure instead of the conventional data and/or instruction cache.

[0056] In general, in one preferred embodiment, the framework may include modifications to an executable such that dedicated architecture support, that may target energy reduction with the help of static information provided by the framework, can be leveraged.

[0057] Furthermore, the method may include, but is not limited to, performance-efficient relocation techniques of memory and branch addresses in an executable.

[0058] As a result, processors using this framework require less energy to operate. Individual instructions may consume less power to execute. Furthermore, the framework is applicable to processors having different architectures and is therefore scalable.

[0059] The invention can be used to save energy on any type of device that includes a processor. For example, the invention can be used to save energy on personal computers, devices containing embedded controllers, sensor networks, network appliances, hand-held devices, cellular telephones, and emerging applications based on other device technologies.

[0060] Unless otherwise defined, all technical and scientific terms used herein have the same meaning as commonly understood by one of ordinary skill in the art to which this invention belongs. Although methods and materials similar or equivalent to those described herein can be used in practice or testing of the present invention, suitable methods and materials are described below. In addition, the materials, methods, and examples are illustrative only and not intended to be limiting.

[0061] Other features and advantages of the invention will become apparent from the following description, including the claims and drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0062] **FIG. 1** is a block diagram showing the relationship between a source-level compiler, instruction set architecture, and microarchitecture in a processor.

[0063] **FIG. 2** is a block diagram showing the relationship between the executable-level re-compiler, conventional source-level compiler, instruction set architecture, and microarchitecture in a processor.

[0064] **FIG. 3** is a flowchart showing a possible embodiment of an executable-level re-compiler according to the present invention and possible interfacing with source-level compilers.

[0065] **FIG. 4** is a block diagram of an instruction memory system that contains the added extensions for Hotblock access to achieve energy reductions based on compiler-architecture interaction and selective mapping between the L1 I-Cache and the Hotblock cache.

[0066] **FIG. 5** is a flowchart showing an example of a procedure as seen and analyzed by the compiler, in one embodiment, to achieve energy reduction in the instruction memory system.

DESCRIPTION

[0067] In the embodiments described herein, a processor framework uses compile-time information extracted from an executable and the executable is then transformed so as to reduce or enable reduction of energy or power consumption, and/or improve performance in a processor. In one embodiment, a reduction can be achieved without requiring special architecture support. In another embodiment, dedicated architecture support could provide unique optimization opportunities based on compiler-architecture interaction. In one aspect, this includes statically managed energy-efficient access mechanisms, such as in the instruction memory, and energy-efficient compiler-driven parallel instruction issue management techniques. This architectural support may be controlled through either executable compilation or source-level compilation. The executable compilation is a method described in the invention.

[0068] Referring to **FIG. 1**, a compiler **10** is a software system that programs circuitry to translate applications from high-level programming languages (e.g., C, C++, Java) into machine specific sequences of instructions. An instruction set architecture (ISA) **12** is a set of rules that defines the encoding of operations into machine specific instructions. The ISA acts as the interface between compiler **(10)** and the microarchitecture **(14)**. A computer program is a collection of machine level instructions that are executed to perform a desired functionality. Micro-architectural (or architectural) components **14** comprise primarily hardware and/or software that are used during execution of the program. The actual machine can be a microprocessor or any other device that is capable of executing instructions that conform to the encoding defined in the ISA.

[0069] Compile-time refers to the time during which the program is translated from a high-level programming language into a machine-specific stream of instructions, and it is not part of execution or runtime. Runtime is the time it takes to execute translated machine instructions on the machine. Compilation is typically performed on a different host machine than execution and it is done on the source program. Energy consumed during compilation does not therefore contribute to the energy consumption of the device targeted.

[0070] Executable-level re-compilation is a process that uses an application's executable as input and performs analysis and modifications at the executable-level. Executable-level compilation is similar to source-level compilation in that it is performed on a different host machine than execution and that during compilation one does not consume energy in the device targeted with the compiled application.

[0071] Referring to **FIG. 2**, an executable-level re-compiler or executable-level compiler **24** is a software system that programs circuitry to translate applications from executable-level into machine-specific sequences of instructions **26**. Components **26** and **28** are similar to the components **12** and **14** in a conventional compiler system as shown in **FIG. 1**.

[0072] Scanning an Executable

[0073] In a preferred embodiment, a module provides a simple interface for extracting program information from a program executable—refer to **34** in **FIG. 3**. Apart from the instructions and static data, section and symbolic information may also be extracted. This could be information that is available in an executable as part of the instruction stream or that is inserted during source-level compilation to facilitate optimizations at the time of executable-level compilation.

[0074] There are a handful of executable file formats used in the industry today, such as ELF, COFF and PECOFF. For example, ARM-ELF defines ARM-specific extensions to the ELF (Executable and Linking Format) standard. In a preferred embodiment, such a module would provide a common interface for accessing executable files in some of these different formats. In one embodiment of this invention, these extensions have been supported.

[0075] Disambiguating an Executable

[0076] In one embodiment of this invention the executable file is analyzed by a binary scanning tool in order to gain information about the various sections and any other symbolic information that may be available. This information is used in order to create a version of the program based on an energy-focused Binary Intermediate Format or BIF, shown as **36** in **FIG. 3**.

[0077] BIF is an abstract representation that facilitates easy retargetability of the analysis and transformation techniques to different instruction sets. It provides necessary information organized in inter-related structures to help perform sophisticated program analyses and transformations.

[0078] Once program analyses and optimizations have been run, the optimized BIF object can be converted back into an executable of the same type as the original one or possibly another type with a different instruction set—see **42** in **FIG. 3**.

[0079] In general, the BIF object may be made up of many procedures or similar program constructs. These procedures may contain a graph of basic blocks or super-blocks, their edges denoting the control flow of the program.

[0080] The basic blocks contain the instructions that are executed in the application. Various information, such as registers, immediates, instruction type, addressing types, condition flags, energy-efficient execution/strength related conditions, etc., may be included about each instruction.

[0081] Once the BIF object has been created, the generated Control Flow Graph (CFG) of the program is subject to many different types of analyses to collect further static program information, shown as **52** in **FIG. 3**. During later analysis, other types of representations may be constructed to enable for example detection of program dependencies and ease program transformations. These representations might include dependence trees, dominator trees, and representations that enable removal of false dependencies, e.g., single state assignment form.

[0082] One of the initial analyses may be the loop analysis. This analysis may use a dominator algorithm to find all of the natural loops in the program. Additionally, induction variables, loop footers, step functions, loop-carried dependencies, early exit points, and memory access stride patterns may be identified and the BIF object updated with that information.

[0083] The idea of criticality may now be introduced. Criticality is used to estimate which of the program blocks are more critical to optimize. For example, blocks in loops are often more critical than blocks outside loops. In one embodiment, blocks in loops are assigned an initial local criticality and a global weight for example based on the depth of the loop they exist in, the procedure context in which they might be used, their probability of being run based on static branch prediction, their contribution to overall energy consumption, and additional heuristics.

[0084] With all of the local criticalities set, a global pass may be run to create a final global criticality value for each of the basic blocks in the program. The blocks in the procedures are updated on the basis of the criticalities of the blocks that call their respective procedures. Once this global criticality is set, a weight analysis may be run. This analysis assigns blocks in a program a global weight.

[0085] In one embodiment, weight may be based on the procedure it is in and its static probability of the block being run, assuming a start point at the beginning of its procedure.

[0086] After a BIF object is created that contains disambiguated symbolic information and the CFG, other analyses are run. **FIG. 3** shows general analyses **52** which may be run on a BIF object before power-specific analyses **38** are run. Along with the specific analyses, the ability to use abstract symbolic execution **40** also exists.

[0087] Abstract execution may be used during various disambiguation phases. It may be used, but not limited to, in forward mode as a fast functional simulation kernel, or as a back-tracking tool on the CFG to disambiguate ambiguous program structures such as call points and branches that have register operands.

[0088] Referring to **FIG. 3**, the executable re-compiler **30** can be integrated with various source-level compilers in the front-end **48** that have source files **46** as inputs. The output of the front-end system is in various binary executable formats, such as used in input **32**. The backend system **50** is responsible for transforming the executable input into an energy-aware binary **44** through a number of analysis and transformation passes.

[0089] Energy-Efficient Modifications Requiring no Architecture Support

[0090] In general, the framework could be used without requiring any modifications in a processor hardware.

[0091] In one embodiment, after a BIF object is created that may contain some or all of the disambiguated symbolic information and the CFG, energy-related analyses may be run. Along with the specific analyses, the ability to use abstract symbolic execution, for example, to determine data memory access patterns, can exist.

[0092] In one aspect, these analyses may include local and global analyses to perform instruction scheduling. In one aspect, the blocks that have been determined as consuming more energy, during criticality analyses, may be optimized in favor of blocks that are determined as less critical. As such, energy consumption may be increased on blocks that

are determined as less critical or such blocks may not be targeted at all. Such increase in fact is to reduce the consumption elsewhere in blocks that are more critical to overall energy consumption.

[0093] Energy-focused instruction scheduling aims to arrange instructions or modify sequences of instructions to reduce energy. This could include rearranging instructions such that pipeline bubbles between dependent instructions are removed, such that instructions with lower energy/power consumption are used whenever possible, and such that architecture support that is present in a processor and that could reduce energy consumption can be better utilized.

[0094] In one embodiment, an example of such existing architecture support includes tag reduction techniques, such as used for example in an ARM 10 processor, where tag access can be removed to save energy whenever the instruction or data access is found in the same cache line as the previous memory access. Instruction scheduling may optimize for such scenarios.

[0095] Additional techniques can take into consideration branch folding scenarios to remove branching related overhead. Branch folding is a mechanism that requires folding the condition of a branch into a following instruction and removing the branch from the processor pipeline. If the branch is predicted correctly, typically determined when the following instruction reaches the execute stage, the branch effectively takes zero cycles to execute.

[0096] Other techniques group memory accesses together from several iterations to utilize more energy-efficient load-store multiple operations instead of individual loads and stores, such as present in some instruction sets.

[0097] Additional techniques may include energy-focused software pipelining and loop transformations. Such techniques can be applied to remove pipeline bubbles between dependent instructions with issue-distances larger than one cycle.

[0098] In such analyses, critical loop iterations may be transformed such that instructions from various loop iterations are pipelined or interleaved to reduce energy and/or improve performance. Loop iterations could be rearranged in a manner that reduces energy. False dependencies might be removed both within and across iterations. This can be accomplished with software register renaming, occasional loop-unrolling, hyperblock formation to eliminate non-critical control-flow, and other techniques. These techniques may be followed by a register allocation pass.

[0099] In one aspect, such transformations may be combined with an energy-focused global instruction scheduling, that involves code motion, to further reduce energy consumption on blocks that are on critical paths.

[0100] A global instruction scheduling rearranges instruction order such that estimated total energy consumption is reduced. It may increase energy consumption on non-critical paths in order to reduce energy on critical execution paths.

[0101] In one embodiment of the executable-level compilation for energy efficiency that has been implemented, targeting a leading 32-bit embedded microprocessor design from ARM, significant energy reduction could be achieved without requiring added architecture support. This reduction has been noticed for a wide range of important applications including video, audio, graphics, security, office automation, compressions and decompression.

[0102] Energy Optimizations Using Architecture Support

[0103] A number of analyses and optimizations could be performed targeting reduction of energy consumption based on dedicated architectural support that can be added to a processor.

[0104] In one aspect this may include, but it is not limited to, techniques that leverage statically managed energy-efficient hardware. Next, we present a number of such mechanisms. The framework is however not limited to only those that are described here. The compiler support can be implemented either through source-level or with executable-level re-compilation. The approach is not limited to using executable compilation.

[0105] On-Demand Static Issue (OSI) Processing and its Control

[0106] In one embodiment, energy consumption may be reduced by determining at compile time which instructions can be executed in parallel, without requiring special hardware disambiguation techniques, and adding at compile-time control information into the instruction stream ahead of their execution. In one aspect, such an approach can be integrated with other energy optimizations, resulting in the control overhead being amortized across many different, for example energy-related, optimizations in a processor.

[0107] In a superscalar processor design, parallelism extraction and scheduling is performed with hardware techniques; while superscalars can provide good improvement in performance, such designs are energy inefficient due to the runtime hardware support required to find and control parallel execution. Such hardware support is required, for example, to extract parallelism from instructions, hardware renaming, and reordering of instructions. Embedded processor manufacturers are reluctant to add parallelism via such an approach to their designs. This is because the increased complexity of the added hardware offsets the benefits of improved performance, due to the increase in load capacitances that increase power consumption.

[0108] By contrast, the parallelism method and processing system that is provided in this invention is on-demand and controlled fully statically. It is referred to as on-demand static issue (OSI) parallelism in the remaining text.

[0109] In comparison with compiler-managed techniques such as VLIW (Very Large Instruction Word) architectures, the OSI technique could be used selectively in code sequences where there is significant instruction-level parallelism and where the overall energy efficiency is estimated to improve, and not applied in codes where there is little parallelism and thus no energy benefit at runtime. As the parallelism is encoded for several instructions ahead of their execution, when parallelism is not exploited, the corresponding execution units can be put instead on standby. A key reason OSI is highly energy-efficient is because it has a minimal impact on the hardware. Moreover, when there is no parallelism in the program, the approach does not impact on the processors energy efficiency. The OSI method is selectively used to achieve as much energy reduction as possible.

[0110] The OSI solution does not affect significantly load capacitances in processor pipelines, as most of the support required is in the compiler where no runtime-related energy is consumed. The required hardware support could include added read and write ports on the register file and the ability to decode and fetch multiple instructions per cycle.

[0111] OSI does not require a fixed multi-issue instruction format similar to a VLIW design. In fact, a processor that supports OSI is backward compatible with codes that do not expose OSI parallelism: another key practical benefit.

[0112] The control bits required for OSI depend on the level of parallelism targeted, but could be as few as one bit per group of instructions specified to be issued in parallel. In a 2-way OSI embodiment it would be sufficient to add one bit per group, or three bits in a six-instruction long basic block. Other types of encodings are however also possible.

[0113] In a preferred embodiment of the invention, the control bits required by OSI could be combined with control bits required for other optimizations.

[0114] The control can be encapsulated for an instruction sequence such as a basic block or super block. This way, static control overhead due to the code dilution is optimized across many other, for example energy related, beneficial optimizations.

[0115] The OSI parallelism approach would also be applicable to existing processor designs as it can be an add-on component. A processor that would be equipped with such a feature would be backward compatible with compilers that do not expose parallelism at compile time; that is, previously compiled executables could also be run unmodified on such a processor. This feature makes OSI an easy upgrade on simple embedded designs where backward compatibility is often required to be preserved due to market constraints.

[0116] In one possible embodiment that we present next, the OSI-related compile-time parallelism analysis is broken up into three different schemes. The first scheme finds the parallelism that is already present in a basic block. The second global scheme reschedules the instructions in several basic blocks globally, to bring out more parallelism. A third approach, is based on a basic block and hyperblock-level software-pipelining technique to expose more parallelism in critical loops. Codes that have extensive control-flow and complicated do-across loop-structure, such as having early exit points and loop-carried dependencies, can also be handled.

[0117] The first, simplest, parallelism scheme traverses the program, performing the analysis on each block that has already been annotated as a critical block. Once a critical block is found, the analysis starts with the first two instructions, in a 2-way OSI embodiment. The two instructions are first checked to see if they could possibly run in parallel given the restrictions of the second OSI pipeline. The types of operations supported in the second pipeline are implementation or embodiment specific.

[0118] For simplicity of describing the techniques we assume 2-way OSI pipelines, however, this is not a limitation—any number of additional execution units and with various capabilities can be added, depending on application and vertical markets targeted.

[0119] The checks that are made for the second pipeline in a possible embodiment are as follows. In one of its aspects, instructions cannot be run in parallel only if: the first instruction is not conditional and the second instruction is (the status bits could be affected); either of the instructions contain the Program Counter; either of the instructions are branch instructions; neither of the instructions is of type Data Processing; both instructions are multiply instructions; one instruction is a multiply and the other is not a data processing instruction.

[0120] These checks reflect a design where the second OSI pipeline only has limited capabilities to execute certain type of arithmetic a logical instructions. This is not a limitation: it is the choice of the designer to decide what capabilities to have in each pipeline, or how many pipelines to add.

[0121] There are also special instructions in different architectures that might not be run in parallel (e.g., saturated add/sub in ARM (QADD,QSUB), state changing instructions in ARM (MSR, MRS)).

[0122] If the instructions can be run in parallel, given the hardware constraints, they are then checked for inter-instruction dependencies. This dependency analysis starts by checking the second instruction for dependencies on the first instruction. Read-After-Write (RAW) hazards as well as Write-After-Write (WAW or output) and Write-After-Read (WAR or false) hazards are dependencies that typically prevent the instructions from being run in parallel. Software register-renaming techniques in the compiler can be used to eliminate both inter- and intra-iterations-based output and false dependencies.

[0123] If the pair of instructions passes both the pipeline restrictions and the dependency tests, there is one final check that can be made. This is another dependency check, only this time, the next instruction in the basic block, i.e., the 3rd instruction in this example, is checked for dependencies with the first. This check is made because if there is a dependency between the first and the third, there might not always be a speedup if the first and second were run in parallel. If all checks are passed, the first and second instructions are annotated to be run in parallel and at code generation, the necessary control bits are inserted into the control instructions.

[0124] This process is repeated until all the instructions in the basic block have been analyzed with their following instruction.

[0125] A second scheme for OSI parallelism, instead of taking the instructions in the order the original performance compiler put them, it reschedules the instructions in a manner to bring out more parallelism. This scheme can be complemented with a third approach based on a software pipelining, to expose more parallelism in loops. There are many other ways parallelism can be exposed and the embodiment described here is not intended to be limiting.

[0126] The OSI instruction rescheduling pass builds a dependency graph for the instructions in the basic block. Along with this dependency graph, each instruction is checked against all of the following instructions and noted whether or not it could be run in parallel with them. Once this is complete, a modified scheduling algorithm is used to reschedule the instructions.

[0127] The rescheduling algorithm used starts with the instructions that have no incoming edges (instructions that don't depend on any other instructions). If there is more than one instruction in this group of candidates, the instruction with the earliest original place in the basic block is chosen to be scheduled. After an instruction is scheduled, all of the outgoing edges (instructions that depend on the instruction that was scheduled last) are possible candidates. These possible candidates are only deemed candidates if all of their incoming edges have already been scheduled.

[0128] If there is more than one instruction that could be run in parallel with the previously scheduled instruction, the instruction with the earliest original placement is selected.

[0129] There is an additional step that may be necessary in instruction sets that support conditional execution. To ensure correctness, the instruction that alters the status bits, which the conditionally executed instructions depend on, is always scheduled as late as possible. This ensures that the conditionally executed instructions that follow will have the correct status bits set. This ensures that the conditionally executed instructions that follow will have the correct status bits set.

[0130] Compiler-Inserted Control to Reduce Energy in the Instruction Memory

[0131] This part of the invention relates to compiler-architecture interaction based approach to reduce energy consumption in the instruction memory system.

[0132] An optimization may target reduction of energy in the instruction memory by directing instruction memory accesses to other more energy-efficient memory structures than the conventional memory structures.

[0133] This control can be incorporated by adding one or more static control bits per controlled sequences of instructions. In fact, a scheme that uses one bit per controlled sequence is possible.

[0134] In one embodiment, shown in **FIG. 4**, there are two memory structures used in the memory architecture—both are compiler controlled. A first structure is a small cache or an SRAM memory or a partition of the first level cache, called the Hotblock **82**; a second structure is equivalent to a conventional first level cache **98**, and requires either a conventional lookup **74** or could avoid tag-lookup **74** if it is accessing data in the same cache line as in the previous access, shown as mechanism **80** and **96**. In a different embodiment the first level cache could be a Ram-Tag based cache instead of the CAM-tag based shown in the figure. A key difference between the two cache types is that in a CAM-based design the tag-check is performed with a Content Addressable Memory (CAM) structure where tags in each way in a set are compared within the CAM after which the correct word-line is selected if the operation results in a cache hit. By contrast, the Ram-based tag structure would require indexing that is performed after word-line decoding from the input address bits. This indexing is performed typically in parallel for the data-array and tag-array.

[0135] Another embodiment may have multiple levels of caches and the addressing of the caches may be physical or virtual.

[0136] In one aspect, the compiler selects to map sequences of instructions, such as a basic block, to either of these structures. When a compiler maps sequences of instructions to the smaller Hotblock structure **82**, enabling signal **90**, instruction memory related energy consumption can be significantly reduced due to the reduction of access energy. This reduction is achieved because the Hotblock cache has a much smaller access energy compared to the first level cache. Clearly, the more accesses are mapped to the Hotblock cache correctly, that is, without causing a Hotblock cache miss, the greater the improvement in energy efficiency.

[0137] Moreover, cell and bitline related leakage energy could be reduced in the first level instruction cache during a Hotblock access by providing support for putting L1 cache lines that are mapped to the Hotblock cache into low-leakage modes.

[0138] The benefit of a compiler-driven strategy for leakage reduction is that it would allow controlling tradeoffs between awakening energy/delay, performance degradation, and leakage power reduction, at a fine granularity and without impacting hardware complexity significantly.

[0139] Inactivity periods are program dependent, as resource demands change as execution proceeds. When inactivity periods are longer and highly predictable at compile-time, we could select to put the circuit blocks in lowest possible leakage power modes. When the inactivity periods are unknown or speculative, we could instead select to use state-preserving approaches (or no leakage control) that, while saving less energy, have lower awakening costs in case of (inactivity period) mispredictions.

[0140] Low leakage modes can be based on gating the power supply **94** on cache lines and/or reduced power supply on cache lines, also called drowsy lines, controlled in block **94**. Other schemes are also possible; the compiler control can be adapted to various circuit-level implementations to address leakage in SRAM cells, as well as control precharging to reduce leakage in the bitlines. Precharging-related leakage energy can be avoided by precharging only the first level cache bank that contains the memory access, instead of all the banks.

[0141] The actual leakage power savings depend on the fraction of drowsy cells compared to gated cells in the L1 data array and other factors such as the time periods during which we can maintain low leakage states in L1 cache lines. The leakage power in the Hotblock cache is small given its small size, as leakage power is at first order proportional to the size of the circuit. Another mechanism used, controls cell and bitline leakage during L1 accesses.

[0142] Bitline leakage is reduced by precharging only the right cache bank whenever the compiler (or the last cache line tag buffer **80**) guarantees intra-line cache accesses. Furthermore, instructions that are placed into low leakage modes during previous Hotblock accesses, would remain in low leakage modes until accessed again.

[0143] With careful compiler orchestration of only mapping the energy critical sequences to the Hotblock structure and reusing the Hotblock across critical loops, a very small cache size can be used. In this embodiment, a cache of 1 Kbyte size provided a sufficient storage for optimizing a wide range of applications studied.

[0144] Independent of the Hotblock cache size, the compiler can restrict accesses to achieve a very high Hotblock hit rate.

[0145] If the Hotblock cache integrates protection related to virtual memory, then address translation from virtual to physical, can be avoided by enabling signal **76**.

[0146] The method in this invention can keep the Hotblock miss rate **92** very low by mapping selectively critical sequences of instructions, estimating the required memory footprint, and managing it at fine granularity with control bits inserted into the instruction stream at fine granularities.

[0147] Furthermore, the control can be integrated with other energy optimizations based similarly on compiler-architecture interaction: any overhead from static control could be therefore amortized across several energy optimization. The negative effects of the control, if any, are more than compensated for by the energy reductions obtained in several critical processor components.

[0148] In the embodiment evaluated we have found that the compiler could limit the miss rate in such a Hotblock structure to below 1%. In one embodiment, this has been demonstrated for a structure as small as 1 KByte size and for a wide range of applications, including SPEC2000, Media, and Mibench benchmarks. These benchmarks represent a cross-section of typical embedded applications from many industries including telecom, security, media, and automotive.

[0149] This can be attributed to selecting at compile time only the very critical blocks and by guaranteeing at compile time that thrashing in the Hotblock structure during critical program sequences such as critical loop iterations do not occur.

[0150] The architecture used can be fully flexible in the way the various structures are ordered. In one preferred embodiment the first level cache **98** can also be accessed directly, depending on compiler information, and it can also be accessed on a miss **94** in the Hotblock structure.

[0151] The compiler would map selected critical blocks, at a fine granularity, to the Hotblock structure and less critical blocks directly to the first level cache.

[0152] In this embodiment, a Hotblock miss is looked up in the first level cache. If the entry is found in the first level cache the corresponding entry in the Hotblock structure is updated.

[0153] A hotblock miss results in a one-cycle penalty. However, with compiler control, the miss rate can be kept very low. Non-critical instructions are directly fetched from the first-level cache avoiding Hotblock misses. The Hotblock cache is reused across instruction sequences such as different loop structures.

[0154] In general, compiler control decides which one of the memory structures is accessed and for which selected instruction sequences, such that overall energy consumption is optimized. The compiler analysis ranks all basic blocks, or larger blocks, in terms of their estimated contribution to the processor's total energy consumption, to determine the memory structure used.

[0155] Referring to **FIG. 5**, key steps of an instruction analysis **100** are shown on a procedure called Proc **57**. Flowchart **112** denotes a CFG with basic blocks including flowchart **114** that denotes Proc **57**. The next flowchart **116** shows basic blocks with detected loop structures including outer-loop **102** and inner-loop **104** and loop-footer **108** for inner-loop. The outer-loop is also shown in flowchart **120** as **118**. As shown, the outer-loop encapsulates the inner-loop. During analysis in flowchart **120**, weight and loop-related criticality is calculated for each basic block. Weight is recalculated after each branch. Depending on embodiment, a weight is attributed depending on the position of the block in the CFG. Criticality in the procedure is added based on loop nest information extracted during analysis on Flowchart **116**. Flowchart **122** shows all blocks updated with weight and criticality by also taking into consideration incoming procedure criticalities such as due to procedure **126** that has an incoming edge to procedure **57**. After this global criticality analysis, the most critical basic blocks **110** are determined as shown in flowchart **124**. These are the blocks that are ultimately mapped to the Hotblock cache. If their footprint exceeds the available space, for example due to fragmentation in the cache, the blocks are mapped in order of their criticality. Fragmentation and other aspects such as cache line size and cache associativity can be taken into consideration during the mapping to avoid overflowing the available space in the Hotblock cache and causing unwanted cache misses. This analysis shows an example with one Hotblock cache. Other embodiments can be based on several Hotblock caches and of different sizes. In such cases, the analysis would map the most critical blocks starting from the smallest Hotblock cache and so on. This organization is not intended to be limiting. Any hierarchical organization of memory structures can be mapped to blocks in the same fashion based on a compiler-selected criticality (or optimization) criterion.

[0156] Other Compiler-Inserted Control for Energy Reduction

[0157] Furthermore, data memory accesses could be directed similarly to more energy efficient access mechanisms to avoid tag-lookups and associative data-array accesses, or to access more energy-efficient memory structures such as a smaller cache or a small cache partition. This may include determining the memory access strides or the criticality of an instruction in an instruction stream. Control information could be added, encoding such sequences of instructions, to enable streamlining a data memory access.

[0158] Unnecessary register file accesses can be avoided by exposing the register access to the compiler and by avoiding register access for registers that are short-lived by accessing the value directly from added buffers in a design, such as accumulators and bypass latches.

[0159] In one aspect, the approach enables replacing dynamic bypass networks in processors, with compiler controlled explicit bypassing.

[0160] In general, in one aspect, dynamic branch prediction techniques could be complemented or replaced by energy-focused static branch prediction. Additionally, compiler-managed prefetching or prefetching hints could be incorporated to reduce cache related energy consumption in designs that otherwise aggressively use prefetching.

[0161] Encapsulating Compiler Inserted Control for Several Optimizations

[0162] All these mentioned optimizations and others not described in this embodiment could be integrated into a processor-wide solution and controlled with one or more

control instructions per sequence of instructions. This would reduce the control overhead as the code-dilution overhead would be amortized across many different energy optimizations.

[0163] In one aspect, the control bits could be added as data to an executable. In one aspect, this data can be accessed through a small dedicated hardware mechanism that would associate branch targets to addresses where control bits are located in the data memory, for example through a hashing mechanism, to fetch the corresponding control bits. A static decode unit would decode these bits and generate the control signals for the associated sequence of instructions.

[0164] During execution, a sequencing mechanism would control both processor resources and select appropriate energy efficient access mechanisms corresponding to the encodings decoded from the compiler-generated control bits.

[0165] The control information can also be incorporated as regular extensions to the instruction set or as co-processor instructions.

[0166] Minimal added pre-decoding logic could be used to remove the control instructions before entering processor pipelines.

[0167] Transformations to the Binary and Relocation of Executable

[0168] Modifications performed on the executable may require relocation of procedure call addresses and memory operations as well as code and other sequences in an executable.

[0169] A method in this invention provides such a relocation in a performance-efficient way in case ambiguous branches or memory accesses exist in the executable. Ambiguous branches or memory accesses may refer to situations when addresses are not known at compile time.

[0170] The challenge is that ambiguous call-points may refer to any procedure in the program; thus if the start point of a procedure is changed in the executable the ambiguous call point may execute incorrectly after relocation is performed.

[0171] In one embodiment, a solution is provided based on the observation that if the original program addresses in the executable of these ambiguous instructions are kept unmodified and all procedures entry points are kept unmodified after relocation, the relocation can be easily accomplished. This can be achieved easily by moving some portion of a procedure to the end of the executable.

[0172] A sample implementation, including the optimizations outlined and performed at executable-level, achieves energy reduction of 26% to 68% when implemented to extend a leading embedded low-power processor.

OTHER EMBODIMENTS

[0173] The invention is not limited to the specific embodiments described herein. The invention is not limited to reducing energy consumption. It could also be used, but is not limited to, to improve performance or improve security.

[0174] Other types of compiler analyses and/or architecture support may be used. The invention may be applied to control any appropriate component of a processor. The optimizations and/or analyses may be performed in a different order and combined with other techniques, in other embodiments. There are different forms of energy consumption, for example such as dynamic or leakage power. The invention is not limited to one source of energy consumption; compiler control for both leakage and dynamic energy can be accomplished as described.

[0175] Other embodiments not described herein are also within the scope of the following claims.

What is claimed is:

1. A method, for use in a processor context, comprising of:

analyzing a program executable at compile time;

creating an intermediate program representation of the executable;

analyzing the intermediate representation;

performing a transformation on the intermediate representation such that a design objective is optimized;

generating optimized program executable from the intermediate representation;

2. The method of claim 1, further comprising of transforming an executable to reduce energy consumption in a processor.

3. The method of claim 1, further comprising of transforming an executable to improve performance in a processor.

4. The method of claim 1, further comprising of incorporating information related to energy consumption into an intermediate program representation of an executable.

5. The method of claim 2, wherein an instruction sequence of one or more instructions is replaced with another instruction sequence of one or more instructions to reduce energy.

6. A method, for use in an application or processor context, comprising of:

analyzing a program executable at compile time;

creating an intermediate program representation of the executable;

analyzing the intermediate representation;

estimating energy consumption in a processor for an instruction sequence of one or more instructions in an application.

7. The method of claim 6, further comprising of generating an output to visualize energy consumption in an application component.

8. The method of claim 6, further comprising of identifying an instruction sequence of one or more instructions most likely to contribute significantly to energy consumption.

9. The method of claim 6, further comprising of:

obtaining runtime statistics about an application and its activity in a processor through simulation;

combining simulation information and compile-time information to obtain energy estimations of one or more instructions.

10. A method wherein:

control information is added at compile time related to execution of a sequence of instructions, of one or more instructions, and controlling plural architectural components during execution;

control information related to a sequence of instructions, of one or more instructions, encoded into a processor instruction.

11. The method of claim 10 wherein a control information is inserted targeting energy reduction in a processor.

12. The method of claim 10 wherein control information is encoded into an instruction part of an instruction set architecture.

13. The method of claim 10 wherein control information is encoded into a coprocessor instruction.

14. A processor framework of claim 10 wherein control instructions are inserted for selected instruction sequences and removed at runtime before entering a processor pipeline.

15. The method of claim 10 wherein control information is inserted as program data.

16. A method wherein control information inserted at compile time comprises of:

which instructions in an instruction sequence are to be issued in parallel at runtime;

additional information related to one or more of the following aspects:

which instruction memory structure an instruction sequence should be fetched from,

length of the instruction sequence,

branch prediction mechanism, and

data memory access mechanism.

17. A processor framework of claim 10 comprising of a compiler-encoded instruction-level parallelism approach wherein instructions in a sequence of instructions are issued in parallel at runtime on compile-time selected blocks of instructions.

18. A method of claim 16 wherein coprocessor instructions encode instruction level parallelism.

19. A method, wherein control information is inserted at compile time to determine explicitly which instruction memory structure an instruction sequence should be fetched from.

20. The processor framework of claim 19 comprised of:

one or more Hotblock instruction memory structures exposed to the compiler;

a first level memory structure exposed and controlled by a compiler;

21. An electronics system that implements claim 20.

22. A processor framework of claim 20 where leakage energy reduction is optimized in a memory structure during a time a Hotblock structure is active.

* * * * *