Architectural Approaches for Dynamic Translation and Reconfiguration

Brian F. Veale[*], John K. Antonio[*], and Monte P. Tull[†]
[*]School of Computer Science [†]School of Electrical and Computer Engineering
University of Oklahoma
Norman, OK 73019
E-mail: {veale, antonio, tull}@ou.edu

***Abstract* - A microprocessor taxonomy is introduced based on whether: (1) the hardware is static or reconfigurable and (2) the code translation process is static or dynamic. The IBM DAISY and Transmeta Crusoe™ microprocessors are reviewed. These static hardware microprocessors support a dynamic translation process to execute programs originally compiled for the PowerPC and Intel® X86 microprocessors, respectively. Inspired by features from both the DAISY and Crusoe™ microprocessors, a conceptual design of a dynamically reconfigurable microprocessor is given. Driven by the results of a preliminary study, a specific approach to designing a reconfigurable microprocessor is presented. As a part of this approach, the concept of partitioning the instruction set of a microprocessor in order to support an application, instead of partitioning the functionality of the application, is developed.**

I. INTRODUCTION

Microprocessor hardware can be divided into two main categories:

1. microprocessors implemented in static hardware; and
2. microprocessor implementations that include reconfigurable hardware.

In a microprocessor implemented in static hardware, the circuitry is fixed and implements the original set of operations for which it was fabricated. However, in a microprocessor implemented using reconfigurable hardware, the operations performed by the reconfigurable circuitry can be changed after fabrication by configuring the reconfigurable hardware. A microprocessor based on reconfigurable hardware can be partially or completely implemented in reconfigurable circuitry, e.g., only the circuitry that performs arithmetic operations might be implemented using reconfigurable circuitry.

An overview of a microprocessor taxonomy is illustrated in Figure 1. In addition to categorizing the type of hardware used to implement the microprocessor, distinction is made in how code is translated, i.e., statically or dynamically. Examples associated with all but one of the categories are shown on the figure.

A. Static Microprocessors

In a static microprocessor, the instruction set that can be executed is fixed and the architecture of the underlying hardware is fixed. Examples of static microprocessors include the Intel® X86 family of microprocessors [1] and the PowerPC microprocessor [2].

The static translation process, which is the typical code development and execution process for static microprocessors, is shown in Figure 2. The source code is constructed using a high-level language, e.g., C++. The compilation process takes in source code and produces machine code for the target microprocessor. In the model of Figure 2, note that the process of translating source code into machine code occurs before execution begins on the static microprocessor.

In addition to the typical static translation process, there exist static microprocessors that perform the translation process dynamically at the same time that execution of the machine code occurs. The generic code development and execution process for a microprocessor that performs dynamic translation is shown in Figure 3.

In dynamic translation, as shown in Figure 3, the source code is developed as before using a high-level language. The compilation process takes in the source code and produces machine code for an initial target microprocessor. This initial target may be associated with an actual physical microprocessor or it may be associated with a virtual microprocessor. (For example, Java source code is initially targeted to binary Java Virtual Machine (JVM) code [3].) The machine code for the initial target microprocessor is re-translated into machine code for the final target microprocessor and optimized. *Re-translation* refers to the process of translating the machine code for the initial target microprocessor into machine code for the final target microprocessor; and *optimization* refers to techniques used to change and re-order the execution of instructions contained in machine code in order to speed-up execution. The re-translation and optimization step is the essence of dynamic translation and can be performed in software or hardware, as illustrated in Figure 1.

This work was supported by the U.S. Department of Education through a GAANN (Graduate Assistance in Areas of National Need) Fellowship.

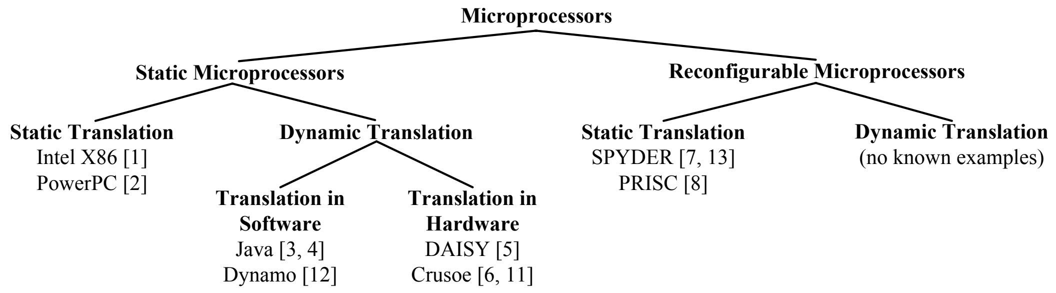


Figure 1. A taxonomy of microprocessors and the translation processes they use.

One example of a system that performs the re-translation and optimization step in software is the JVM [3]. When a Java program is executed on a static microprocessor, the initial machine code, which is called Java bytecode, is re-translated into the machine code for the target microprocessor using the JVM, which is implemented in software [4].

The DAISY (Dynamically Architected Instruction Set from Yorktown) [5] and Crusoe™ [6] microprocessors are examples of static microprocessors that perform the re-translation and optimization step of the dynamic translation process in hardware. In these systems, the source code is not initially compiled for the DAISY or Crusoe™ microprocessor, but for a different static microprocessor. When the initial machine code is executed by DAISY or Crusoe™, it is re-translated into machine code for the DAISY or Crusoe™ microprocessor, which is then executed by the microprocessor [5, 6]. This re-translation is performed in hardware. A main focus of this paper is to overview and compare the DAISY and Crusoe™ systems (Sections II and III).

B. Reconfigurable Microprocessors

In contrast to a static microprocessor, the instruction set and the underlying architecture of a reconfigurable microprocessor can be dynamic. This means that the instruction set and the circuitry implementing particular instructions or functionality of the microprocessor can be changed after fabrication of the microprocessor.

The static translation process, which is the typical code development and execution process for reconfigurable microprocessors, is shown in Figure 4. The source code is constructed using a high-level language. The compilation process takes in source code and produces: (1) machine code for the target microprocessor and (2) a description of functionality that represents a sequence of instructions to be implemented in the reconfigurable hardware to support the machine code. After the compilation process is finished, the synthesis process converts the descriptions of the instructions to be implemented in reconfigurable hardware into binary configuration code for the reconfigurable hardware. In the model of Figure 4, note that the process of translating source code into machine code and binary configuration code occurs before execution begins on a reconfigurable microprocessor. The SPYDER [7] and PRISC [8] microprocessors are examples of reconfigurable microprocessors that use this approach. A review of these microprocessors is presented in Section IV.

Unlike the category of static microprocessors, there are no known examples of a reconfigurable microprocessor that uses a dynamic translation process. At the end of this paper, future work is outlined in the direction of examining dynamically reconfigurable microprocessor architectures.

C. Summary of Microprocessor Taxonomy

For the purpose of this study, microprocessors are implemented in either static or reconfigurable hardware. Two possible translation processes are defined: static and dynamic. In the static translation approach, the source code is compiled before execution on the microprocessor begins. In the dynamic translation approach, initial machine code is re-translated and/or optimized during execution on the microprocessor.

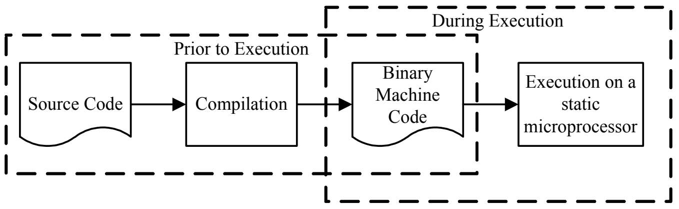


Figure 2. The static translation process for a static microprocessor.

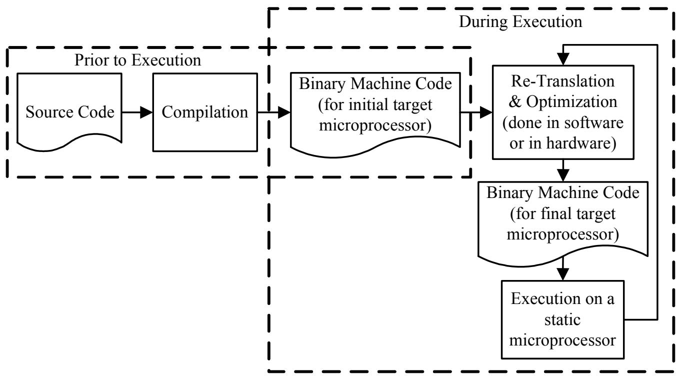

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
Re-Translation & Optimization (done in software or in hardware)
Binary Machine Code (for final target microprocessor)
Execution on a static microprocessor

Figure 3. The dynamic translation process for a static microprocessor.

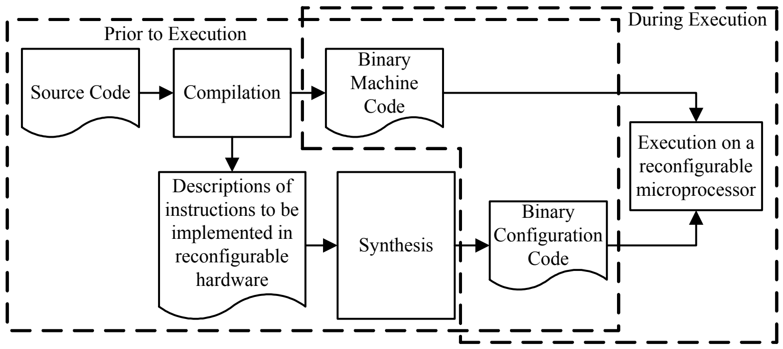

Prior to Execution
During Execution
Source Code
Compilation
Binary Machine Code
Descriptions of instructions to be implemented in reconfigurable hardware
Synthesis
Binary Configuration Code
Execution on a reconfigurable microprocessor

Figure 4. The static translation process for reconfigurable microprocessors.

Microprocessors that perform dynamic translation have the advantage that they can execute machine code that was initially compiled for an existing microprocessor that has an established market share. Microprocessors that perform static translation do not have to perform the re-translation and optimization step found in dynamic translation and therefore may execute faster than a microprocessor that uses dynamic translation to execute the same machine code.

Reconfigurable microprocessors have the potential advantage of being able to dynamically alter their architecture and/or instruction set. However, current technology that supports reconfigurable microprocessors is generally slower than the technology used to create static microprocessors. Even though reconfigurable hardware may be slower, it still holds promise of producing overall performance that is faster than static hardware by strategically and dynamically reconfiguring the architecture to match the currently executing instructions.

This paper describes architectural approaches to support dynamic translation and reconfiguration. Sections II and III provide details on the design of the hardware architectures of the DAISY [5] and Crusoe™ [6] microprocessors, respectively, which are examples of static microprocessors that implement dynamic translation. An alternative approach to the design of reconfigurable microprocessors is proposed and preliminary architectural approaches of a dynamically reconfigurable microprocessor that can use this approach are given in Section V.

II. The IBM DAISY Microprocessor

A. Overview

The DAISY microprocessor [5] is a static microprocessor that has been developed by IBM, which uses the dynamic translation process of Figure 3. The goal of the DAISY microprocessor is to use dynamic translation to provide complete compatibility with the binary machine code of an existing commercial microprocessor [5]. The DAISY microprocessor presented is completely compatible with the machine code of the PowerPC microprocessor. However, the techniques used in the PowerPC version of the DAISY microprocessor could be applied to other microprocessors such as the Intel® X86 and the IBM System/390, as well as virtual microprocessors such as the JVM [5].

A high-level component view of the DAISY microprocessor is shown in Figure 5. The architecture of the DAISY microprocessor is based on a VLIW (Very Long Instruction Word) processor core and is built on top of the PowerPC memory model and register file [5]. The white areas of Figure 5 represent PowerPC components of the system, and the black areas represent the DAISY specific components of the system. Note that there are no PowerPC execution units; all processing is done in the block labeled DAISY Processor Core (VLIW).

In the DAISY microprocessor, instructions are tree-based and implement a multi-way path selection scheme [5]. The flow control model for a tree-based instruction is given in Figure 6. The multi-way path selection scheme allows the dynamic translation process to aggressively re-translate and optimize programs that contain multiple paths of flow and benefit from branch prediction.

Each DAISY VLIW instruction can specify up to sixteen concurrent operations [5]. In the model of Figure 6, each path can consist of any subset of the sixteen operations. The condition codes (ccA, ccB, and ccC) determine the path taken and what instruction is performed next [9].

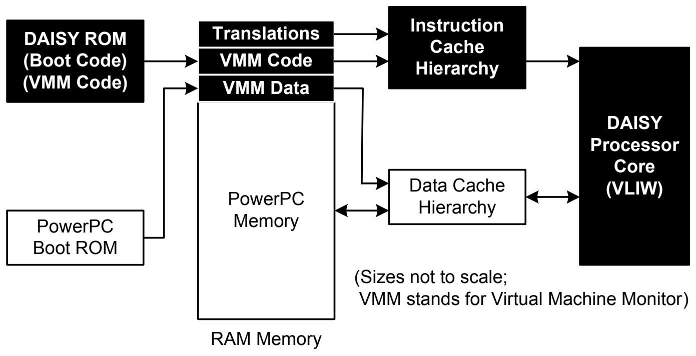
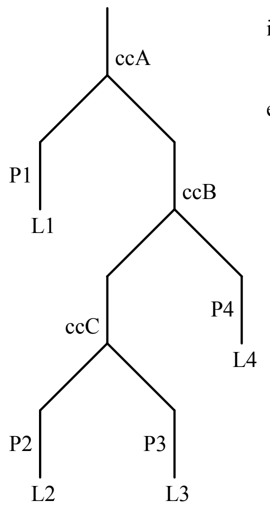

DAISY ROM (Boot Code) (VMM Code)
Translations
VMM Code
VMM Data
Instruction Cache Hierarchy
DAISY Processor Core (VLIW)
PowerPC Memory
Data Cache Hierarchy
PowerPC Boot ROM
(Sizes not to scale; VMM stands for Virtual Machine Monitor)
RAM Memory

Figure 5. The components of a DAISY microprocessor [5].



```
if(ccA == false)
    execute ops on path P1
    branch to L1
else
    if(ccB == true)
        execute ops on path P4
        branch to L4
    else
        if(ccC == false)
            execute ops on path P2
            branch to L2
        else
            execute ops on path P3
            branch to L3
```

Figure 6. The tree-based instruction flow control model and instruction format [9].

The DAISY microprocessor performs the re-translation and optimization step of Figure 3 by performing a re-translation of initial machine code into groups of DAISY instructions (called instruction groups) that are in the form of machine code for the DAISY microprocessor. As execution of machine code on the DAISY microprocessor continues, if previously re-translated instruction groups are encountered frequently, then they are optimized. This process of re-translation and optimization is depicted in Figure 7 [5].

B. Re-translation and Optimization of Binary Machine Code

The DAISY microprocessor uses a Virtual Machine Monitor (VMM), shown in Figure 5, to handle the re-translation process. The VMM also handles control of the microprocessor, including exception handling, and is transparent to the binary machine code of the initial target microprocessor [5].

In the DAISY microprocessor, instruction groups take the form of a tree and are called tree groups. A tree group is a high-level abstraction of a group of VLIW instructions that models the flow of instruction execution (the control path) through a program. This control path defines the tree properties of a tree group. Control paths can only merge on the boundary between tree groups (the transition from one tree group to another). Each of the leaves of the tree corresponds to an exit point in the tree and is called a "tip." By knowing which of the tips was used to exit the tree, the system can determine the path taken through the tree [5].

Tree groups are used as the unit of translation in the re-translation process. When a segment of code is encountered that has already been re-translated, the system merely branches to the corresponding tree group. In this situation, re-translation is not necessary.

If re-translated code is executed frequently, then it is optimized. The goal of the optimization algorithms used in DAISY is to attain a significant level of ILP (Instruction Level Parallelism) with a low overhead cost. The scheduling approaches are adaptive and a function of execution frequency and behavior. The optimizations used in these approaches include copy propagation and load-store telescoping [5]. For a more detailed description of these optimizations, refer to [10].

The process of re-translating code a specified number of times before it is optimized is beneficial to the overall performance of the system for at least two reasons. First, the re-translation process acts as a filter for rarely executed code to keep such code from being optimized. The cost to optimize such code is wasted and will never be regained because the system will not benefit from faster execution of the code in the future. Second, the re-translation process can be used to gather data about how to guide the optimization process.

C. Special Hardware and Control Mechanisms

There are several areas in which special support is provided to make the DAISY microprocessor completely compatible with the binary machine code of the PowerPC microprocessor without encountering performance degradation. Among these areas are exception handling and context switching mechanisms, support for handling register-indirect branches, and being able to detect and handle self-modifying and self-referential program code. A more detailed description of these mechanisms and their underlying hardware support is provided in [10].

D. Summary

The keys to the success of the approaches used in DAISY are that the system performs ILP extraction at execution time [5] and run-time profiling of program code. This results in a high level of performance due to the ability of the microprocessor to dynamically adapt the re-translated instruction code. This is a major improvement over the heuristic and profile-based approaches that static compilers use, that result in trade-offs being considered to improve performance [5].

The performance studies of DAISY, presented in [5], indicate that the filtering out of infrequently executed machine code before it is optimized does not necessarily improve system performance; and that the optimization of DAISY machine code is expensive. Also, it was found that the ILP achieved is directly affected by how tree groups are formed.

III. The Transmeta Crusoe™ Microprocessor

A. Overview

The Crusoe™ microprocessor [6], developed and marketed by Transmeta Corporation, is in the same class of microprocessors as DAISY, i.e., it is a static microprocessor that performs the re-translation and optimization step of the dynamic translation process in hardware. This microprocessor is associated with the same high-level translation process as DAISY, which is illustrated in Figure 3. The goals of the Crusoe™ microprocessor are to be completely compatible with the machine code of the Intel® X86 family of microprocessors [1] and to directly compete with these microprocessors in the marketplace. The Crusoe™

microprocessor achieves these goals with a unique hardware architecture, which includes enhanced support for re-translating X86 machine code into Crusoe™ machine code and executing the resulting machine code [6].

A high-level view of a Crusoe™ based system is shown in Figure 8. Similar to the DAISY, the Crusoe™ microprocessor is based on a VLIW processor core and is built on top of the X86 register file and memory model. A Crusoe™ based system can be divided into four layers: (1) the target application which was initially compiled for an X86 microprocessor; (2) the target operating system (also initially compiled for an X86 microprocessor); (3) the Code Morphing™ process which handles the re-translation and optimization of machine code, the maintenance of re-translated machine code in a translation buffer located in memory, and system control; and (4) the Morph host which is the VLIW processor core of the microprocessor [6].

The Code Morphing™ process of the Crusoe™ microprocessor maintains a translation buffer, as shown in Figure 8, which stores completed re-translations of each X86 instruction. Once instructions are successfully re-translated and segments of instructions are optimized, the resulting machine code is stored in the translation buffer. The resulting machine code (in the translation buffer) is executed by the VLIW processor core. When a previously re-translated instruction is encountered again, the microprocessor can recall the corresponding operation(s) from the buffer and execute them without further re-translation [6].

As X86 machine code is executed on the Crusoe™ microprocessor, if the instruction being executed at any given time has not been re-translated (and does not exist in the translation buffer), then it is re-translated into machine code for the Crusoe™ microprocessor and optimized [6]. This process is the re-translation step of Figure 3 and is shown in Figure 9 for the Crusoe™.

The performance of the Crusoe™ microprocessor comes from the reduction in the amount of hardware in the microprocessor (compared to the Intel® X86 microprocessor) and the caching of re-translated machine code. This results in a possible speedup of the execution of program code and a reduction in the power consumption of the microprocessor [11].

B. Re-Translation and Optimization of Instructions

The Crusoe™ microprocessor uses the Code Morphing™ process, shown in Figure 9, to perform the re-translation process. The Code Morphing™ process also optimizes the resulting machine code and handles control of the system, including exception handling [6]. Just as with the DAISY VMM, this process is transparent to the X86 machine code being executed on the Crusoe™.

The first time an X86 instruction is encountered it is re-translated into a sequence of Crusoe™ operations, as shown in Figure 9. As instructions are re-translated, the different segments of Crusoe™ machine code that are generated are linked together so that they do not branch back to the Code Morphing™ process if the next segment to be executed has already been re-translated. This helps to eliminate most of the branches back to the Code Morphing™ process and serves to enhance the speed of the emulated X86 microprocessor. Once the system has reached a steady state, it is estimated that a re-translation will only be necessary for one in every million X86 instructions executed over the life of a running program [6].

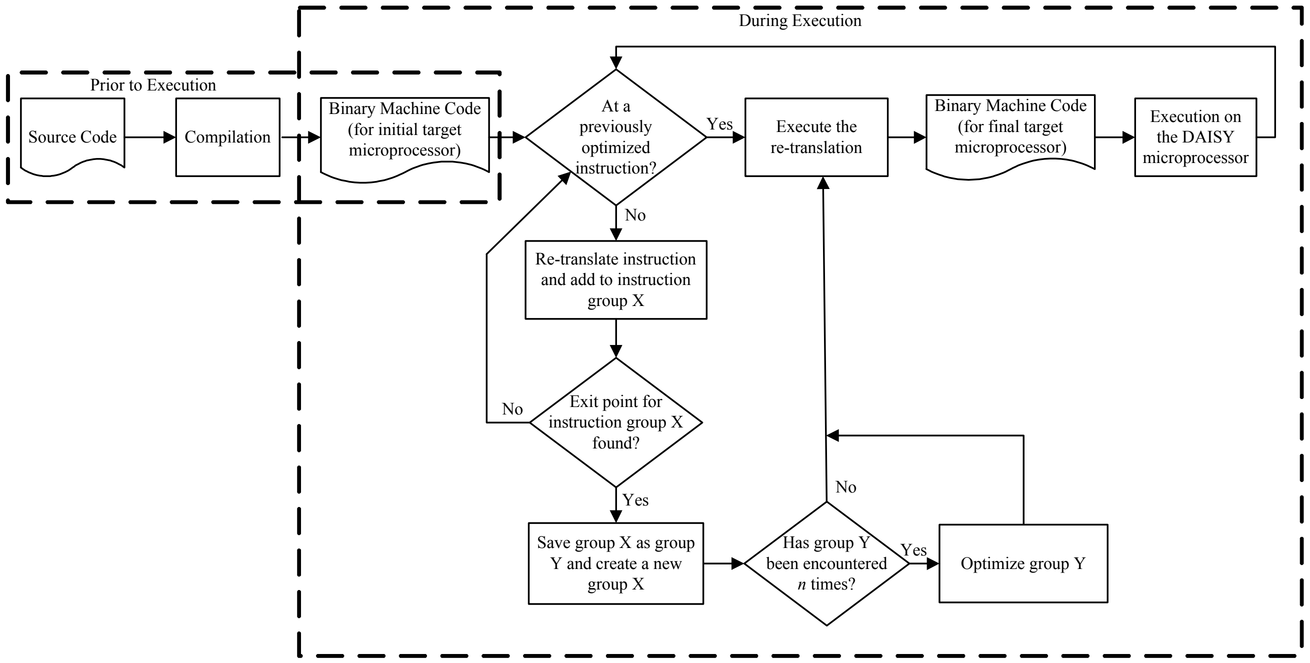

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
At a previously optimized instruction?
Yes
No
Execute the re-translation
Binary Machine Code (for final target microprocessor)
Execution on the DAISY microprocessor
Re-translate instruction and add to instruction group X
Exit point for instruction group X found?
No
Yes
Save group X as group Y and create a new group X
Has group Y been encountered n times?
Yes
No
Optimize group Y

Figure 7. The dynamic translation process used by the DAISY microprocessor derived from [12].

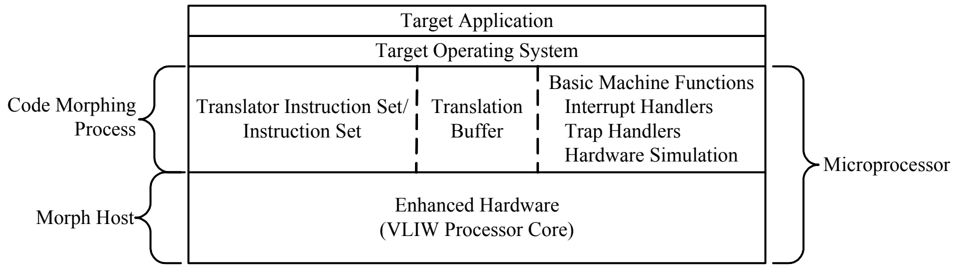


Figure 8. The components of a Crusoe™ based system [6].

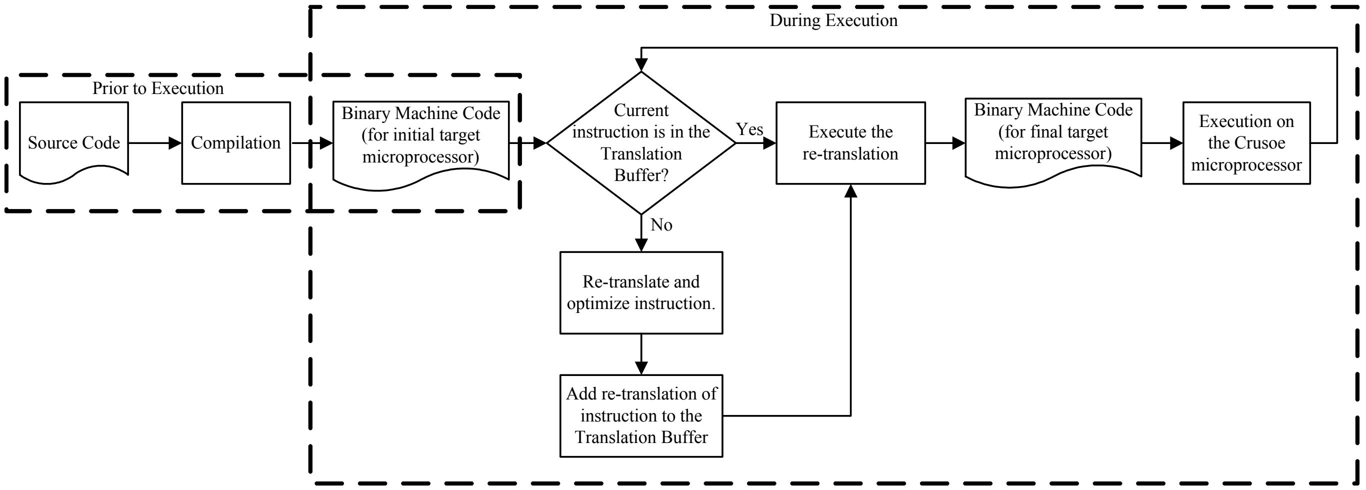

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
Current instruction is in the Translation Buffer?
Yes
No
Execute the re-translation
Binary Machine Code (for final target microprocessor)
Execution on the Crusoe microprocessor
Re-translate and optimize instruction.
Add re-translation of instruction to the Translation Buffer

Figure 9. The dynamic translation process used by the Crusoe™ microprocessor derived from [6].

In addition to re-translating X86 machine code into Crusoe™ operations, the Code Morphing™ process also optimizes the operations in an attempt to speed up the execution of instructions as much as possible. Specific optimizations described in [6] are: speculatively removing the X86 segmentation process; speculatively removing upper boundary memory checks; common sub-expression elimination; speculatively removing commit operations; register renaming; code motion; data aliasing; copy elimination; and the use of alias hardware. These optimizations are performed on a re-translation only if the re-translation is executed frequently, because the time needed to re-translate and optimize infrequently executed instructions is greater than the time required to re-translate and execute the instructions without optimization. Refer to [10] for a comprehensive summary of these optimizations.

C. Commercial Impact

The Transmeta Crusoe™ microprocessor is a successful commercial product. It has gained a share of the Intel® X86-compatible microprocessor market. Transmeta has been successful in marketing the Crusoe™ microprocessor for use in laptop computers and handheld devices. The power requirements of the Crusoe™ microprocessor are 60%-70% less than compatible microprocessors [11]. This has been accomplished by reducing the complexity of the underlying hardware architecture of the microprocessor [6].

IV. The SPYDER and PRISC Microprocessors

SPYDER [7] and PRISC [8] are examples of reconfigurable microprocessors that use the translation process of Figure 4. SPYDER uses reconfigurable resources to implement hardware synthesized specifically for a program to be executed on the processor [7]. A C++ to netlist compiler that creates the binary configuration code used to configure the processor must be run before a program can be executed on SPYDER [13]. Thus, SPYDER requires that source code must be available and recompiled.

PRISC [8] is a reconfigurable processor similar in concept to SPYDER. A main difference between the two is that the reconfigurable resources in PRISC are in the form of execution units connected to the data path of the CPU along with static functional units [8] as shown in Figure 10; SPYDER does not specify static execution units. Because of the static execution units present in PRISC, programs can utilize static integer and floating-point execution units, while this is not possible with the SPYDER processor. For programs to utilize the reconfigurable resources of PRISC, source code must be analyzed by a hardware extraction tool that determines what program code should be executed using the reconfigurable resources [8].

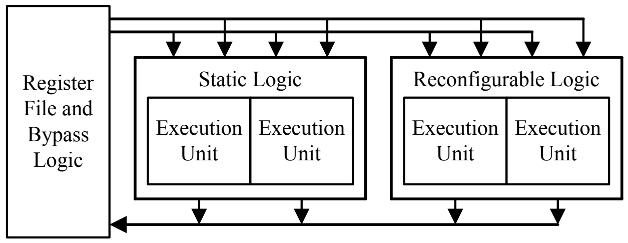


Figure 10. Architecture of the PRISC Microprocessor derived from [8].

The approaches of the SPYDER and PRISC processors represent important steps in applying reconfigurable computing to the realm of general-purpose computing and commercial embedded applications. However, both of these approaches require compilation and analysis at the source-code level, and they do not directly support existing instruction set architectures. Thus, they may lack mainstream viability because they are not legacy-compatible at the machine-code level. This is an important factor, considering the vast amount of legacy software executables and hardware systems that dominate today's market. An alternative approach is presented in the next section.

V. AN ARCHITECTURE FOR A DYNAMICALLY RECONFIGURABLE MICROPROCESSOR

Inspired by the study of the DAISY, Crusoe™, SPYDER, and PRISC microprocessors, a concept of combining dynamic code analysis and reconfigurable technology is presented in this section. The goal is to describe a microprocessor that is compatible with an existing microprocessor and exploits the flexibility of reconfigurable hardware. A high-level architecture of such a microprocessor, that is similar to SPYDER and PRISC, is presented in Figure 11.

A. System Architecture

The microprocessor depicted in Figure 11 combines static execution units and dynamically reconfigurable execution units implemented in reconfigurable hardware to provide a VLIW-based microprocessor that supports configuration context switching. A set of pre-defined execution units can be loaded into a context of the reconfigurable hardware within the microprocessor as needed by the configuration controller, each of which supports a subset of the instructions supported by the instruction set architecture (ISA) of the microprocessor. When an execution unit configuration is loaded into the microprocessor, the data paths and control signals for the execution unit is specified by the configuration controller. This approach allows multiple copies of the same execution unit to be loaded into the microprocessor without any hardware conflicts by assigning the execution units different data paths and control signals. For example, the reconfigurable portion of the microprocessor could be configured with two ALUs and two Load/Store units as depicted in Figure 11.

The configuration controller handles the loading of execution units into the reconfigurable hardware of the microprocessor. The reconfigurable hardware within the microprocessor is capable of supporting multiple contexts (as first suggested in [14]). Reconfigurable hardware contains logic blocks (sometimes referred to as logic cells), which are configured by a string of bits (called a bit-stream) to implement a specified function. In a context switching approach, each bit of the bit-stream is stored in a separate configuration register (or memory) instead of storing the entire bit-stream in a single memory, e.g., SRAM. Therefore, the configuration registers not only specify the current configuration context for each logic block, but also holds other configuration contexts. When the processor wishes to switch to a configuration context that is already present in the configuration registers, it simply has to change which bit of each configuration register is used to specify the configuration.

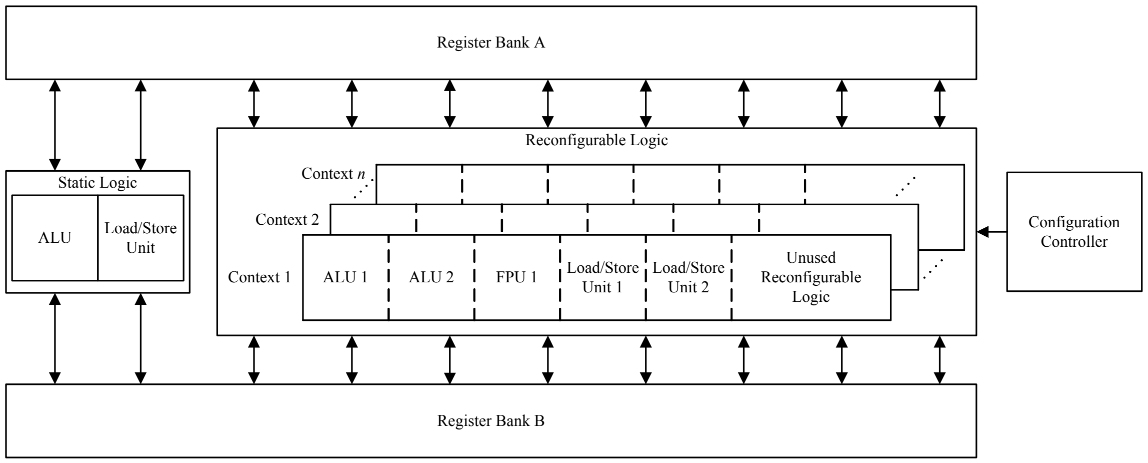


Figure 11. A multiple context reconfigurable microprocessor that uses dynamically reconfigurable and static execution units.

The combination of the configuration registers and the configuration controller can be used to implement a configuration pipeline. A configuration pipeline is similar to an instruction pipeline, and is filled with the configurations that implement different sets of execution units that are needed to execute segments of code. As each set of execution units is needed, the pipeline is advanced and the reconfigurable hardware is reconfigured to implement the units. This approach can speed-up the re-configuration of reconfigurable hardware by decreasing the time required to load a new configuration into the device. This decrease is achieved by removing the need to re-write all of the configuration bits at the point in time when a re-configuration is needed (assuming the desired configuration exists in the configuration register).

Our preliminary studies suggest that there typically exists a subset of instructions (such as those listed in Figure 12 for the Intel® X86 microprocessor) that are frequently executed throughout a given program. These studies examined the execution of the POV-Ray (Persistence of Vision Raytracer) ray tracing program [15] compiled for Linux at the assembly language level one instruction at a time using the Linux system call *ptrace* [16] (e.g., supported in Red Hat Linux 7.3). The *ptrace* utility can trace all code within the program's user space, but cannot follow calls to operating system code. These studies are reported in [10].

The results of our initial studies suggest that there exist instructions that should be implemented statically in hardware in order to optimize the overall execution of the program because these instructions occur frequently and uniformly throughout the execution of a program. In addition to implementing instructions that are frequently executed in static execution units, it is also desirable to design the static execution units so that they can implement any instruction. This makes the microprocessor completely compatible with the entire instruction set architecture it implements for all possible configurations of the reconfigurable hardware. Of course the objective of the configuration process would be to select configurations that best match the instructions currently being executed (i.e., so as to minimize the number of instructions that are sub-optimally matched to the current configuration).

mov	fstp	fucompp	cld
push	fxch	fadd	nop
cmp	fmul	jbe	fsubr
pop	ret	jc	jnc
fld	call	fstcw	jg
test	and	fst	fucomp
jnz	fstsw	jns	leave
add	jmp	fldz	jl
inc	jle	fld1	or
sub	fldcw	fucom	fabs
movzx	dec	shl	frdint
lea	faddp	ja	
jz	xor	fistp	

Figure 12. The fifty most frequently executed instructions in POV-Ray [10].

In the typical translation process for reconfigurable microprocessors, shown in Figure 4, the *functionality* of the program being executed on the microprocessor is distributed amongst a set of configurations. The studies performed in [10] have inspired an alternative approach to the design of reconfigurable microprocessors in which the configurations for the microprocessor each support a subset of the instruction set of the microprocessor. This approach is proposed in the next subsection.

B. Instruction Set Partitioning

One drawback of the approach, taken in SPYDER[7] and PRISC[8], of distributing the functionality of the program being executed on a reconfigurable microprocessor among a set of configurations is that the hardware required to support the program is application specific. Additionally, this approach requires that the program be re-compiled and the functionality of the hardware needed to support the program be extracted from the source code and synthesized into binary configuration code(s) for the reconfigurable microprocessor. Therefore, a reconfigurable microprocessor that uses this translation model is not legacy-compatible at the machine code level.

A different approach is to distribute the functionality of the instruction set supported by the microprocessor among a set of configurations that may or may not overlap (i.e., instruction set partitioning), effectively splitting the instruction set into smaller sets of instructions. This approach moves the definition and synthesis of the functionality supported by the microprocessor off-line; the translation process of such a microprocessor uses a set of pre-defined hardware configurations. This translation process is presented in Figure 13.

In contrast to the translation process for a reconfigurable microprocessor in which the functionality of the program being executed is distributed among a set of configurations (refer to Section I.B), the model of Figure 13 does not generate the binary configuration codes for the reconfigurable microprocessor. Instead of the binary configuration codes supporting functionality of the specific program being executed, they support the instruction set of the microprocessor and are pre-defined. Thus, the same set of binary configuration codes are used for every program executed by the microprocessor, although some programs may not require the existence of all of the binary configuration codes in order to execute.

The translation process may include optimizations performed on the binary machine code in order to improve the performance of the microprocessor by applying techniques such as code re-ordering to reduce the required number of configuration switches. The microprocessor can execute un-optimized code; however, in order to execute a particular instruction, a correct partition must be configured and un-optimized code could result in a high rate of configuration switching.

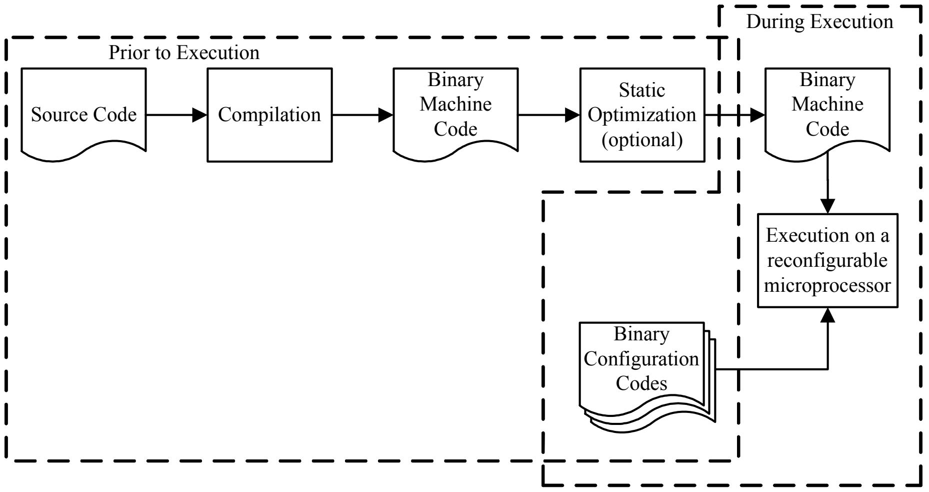

Prior to Execution
During Execution
Source Code
Compilation
Binary Machine Code
Static Optimization (optional)
Binary Machine Code
Execution on a reconfigurable microprocessor
Binary Configuration Codes

Figure 13. A static translation process for reconfigurable microprocessors in which support for the instruction set of the microprocessor is distributed among a set of configurations.

A dynamic translation process for a reconfigurable microprocessor that uses instruction set partitioning extends the static translation process of Figure 13 by utilizing optimizations that benefit from run-time information about the program being executed such as profiling data that indicates the probabilities associated with the targets of control instructions (e.g., branch instructions). Another approach to dynamic translation for a reconfigurable microprocessor that uses instruction set partitioning is to dynamically determine the configurations that support the instruction set during the execution of the program.

There are no known reconfigurable microprocessors that utilize instruction set partitioning. This approach to the design of reconfigurable microprocessors is being investigated and preliminary results of this investigation can be found in [17].

C. The Configuration Process

The configuration process for the proposed microprocessor is split into a static and dynamic analysis of the program. The static analysis occurs at or after compilation time and the dynamic analysis occurs during execution of the program.

The static analysis examines one unit of code (e.g., basic block) of the program at a time and determines the optimal set of execution units to be configured within the microprocessor during execution of the unit of code. The execution units considered by the analysis process include the static execution units and a pre-defined set of execution units that can be loaded into the reconfigurable hardware of the microprocessor. The set of execution units needed for each unit of code within the program can be specified by the static analysis process by adding tags to the end of the instructions that specify the desired configuration of the execution units, which are then interpreted and dynamically implemented by the configuration controller. We note here that the static analysis can be performed directly on binary machine code produced by existing compilers. Thus, this analysis is stand-alone and does not necessarily need to be integrated into existing or new compilers. This implies that this analysis can be viewed as a post-compilation but pre-execution process.

Although useful, the static analysis does not have to be performed; standard legacy code without the addition of configuration tags can be executed directly on the microprocessor. If “non-tagged” code is executed on the microprocessor, then a default configuration is initially implemented in the reconfigurable hardware. The initial configuration used is likely not optimal for the code being executed; however, dynamic analysis commences and more optimal configurations are determined during execution. Dynamic analysis further optimizes the ordering of configurations provided to the configuration controller beyond what the static analysis provides. This ordering is based on predictions of what units of code will be executed in the future. Because the static analysis does not have access to information about the data being processed by the program or the past execution flow of the program, this further optimization can only be done at execution time. To reduce the overhead incurred by reconfiguration of the microprocessor, the dynamic analysis and loading of the configuration registers can be pipelined, as illustrated in Figure 14.

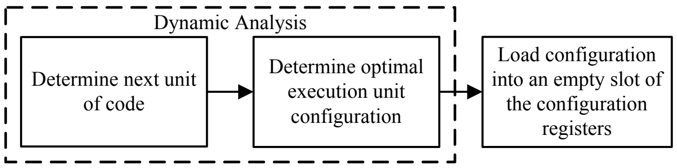

Dynamic Analysis
Determine next unit of code
Determine optimal execution unit configuration
Load configuration into an empty slot of the configuration registers

Figure 14. Pipelining of dynamic analysis and loading of configuration registers.

VI. Conclusions

This paper has introduced a microprocessor taxonomy that classifies microprocessors based on the technology used to implement them (static or reconfigurable), the process that they use to translate machine code and execute instructions, and whether this process is performed in software or hardware. The design and operation of two different static microprocessors that perform dynamic translation of machine code have been presented and compared. A proposed architecture for utilizing dynamically reconfigurable hardware was also discussed.

The two microprocessors reviewed in this paper are the IBM DAISY and the Transmeta Crusoe™. These microprocessors use dynamic translation to execute machine code initially compiled for the PowerPC and Intel® X86 microprocessors, respectively. The design of these two microprocessors and how they perform dynamic translation differ. DAISY is based on a sophisticated VLIW processor core while the Crusoe™ uses a simplified VLIW processor core that has extra hardware support added for speeding up the process of rolling back the state of the emulated microprocessor when an exception occurs. The re-translation, optimization, and scheduling processes are also different between these two microprocessors. DAISY uses a generic and broad approach while the Crusoe™ is Intel® X86 specific and performs specialized optimizations that may only apply to Intel® X86 machine code.

The DAISY and Crusoe™ microprocessors both represent a new direction for microprocessor design. The designers of these microprocessors recognize the reality that for a new microprocessor to be successful in today's market, it should be compatible with an existing instruction set of a microprocessor that has been successful. This is due to the vast amount of legacy software and hardware systems that dominate the market.

Some initial concepts are proposed at the end of the paper related to an architecture that includes dynamically reconfigurable hardware. Such a microprocessor may be able to outperform a static counterpart because the analysis process has the option of executing a unit of code using a particular configuration of the reconfigurable hardware for which it is well matched. The efficient implementation of instruction(s) in reconfigurable hardware could speed-up overall execution.

A new approach to the design of reconfigurable microprocessors in which the instructions of the ISA are distributed amongst a set of configurations for the microprocessor instead of distributing the functionality of the program being executed amongst the configurations is proposed. This approach is not application specific and can be used to develop microprocessors that support both new and legacy ISAs. This approach is currently being investigated and studies dealing with the feasibility of this approach and optimization of code generated for a microprocessor that uses instruction set partitioning are reported in [17].

The goal of the concepts presented at the end of this paper is to minimize the number of reconfigurations required to execute a program while maximizing the amount of ILP achieved. Future work will extend the concepts presented here by formalizing specific approaches to this design problem.

References

[1] *IA-32 Intel Architecture Software Developer's Manual*, Intel Corporation, http://developer.intel.com/design/pentium4/manuals/index2.htm, 2002.

[2] *PowerPC Microprocessor Family: The Programming Environment for 32-Bit Microprocessors*, International Business Machines Corporation, http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256 99600719DF2, 2002.

[3] J. Gosling and H. McGilton, "The Java Language Environment: A White Paper," Sun Microsystems Inc., Mountain View, CA, ftp://ftp.javasoft.com/docs/papers/langenviron-pdf.zip, May 1996.

[4] *The Java HotSpot Virtual Machine Technical White Paper*, Sun Microsystems Inc., Palo Alto, CA, http://wwws.sun.com/software/solaris/java/wp-hotspot/, 2001.

[5] K. Ebcioğlu, E.R. Altman, M. Gschwind, and S. Sathaye, "Dynamic Binary Translation and Optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, June 2001, pp. 529-548.

[6] R.F. Cmelik, D.R. Ditzel, E.J. Kelly, C.B. Hunter, D.A. Laird, M.J. Wing, and G.B. Zyner, "Combining Hardware and Software to Provide an Improved Microprocessor," *US Patent 6,031,992*, Feb. 2000.

[7] C. Iseli and E. Sanchez, "Beyond Superscalar Using FPGAs," *Proceedings of the 1993 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1993, pp. 486-490.

[8] R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 172-180.

[9] K. Ebcioğlu, J. Fritts, S. Kosonocky, M. Gschwind, E.R. Altman, K. Kailas, and T. Bright, "An Eight-Issue Tree-VLIW Processor for Dynamic Binary Translation," *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, 1998, pp. 488-495.

[10] B.F. Veale, J.K. Antonio, and M.P. Tull, "Design and Optimization of Legacy Compatible Microprocessors," Technical Report No. CS-TR-02-002, School of Computer Science, University of Oklahoma, http://www.cs.ou.edu/~veale/pubs/CS-TR-02-002.pdf, Dec. 2002.

[11] A. Klaiber, "The Technology Behind Crusoe™ Processors: Low-Power X86-Compatible Processors Implemented with Code Morphing™ Software," Transmeta Corporation, Santa Clara, CA, http://www.transmeta.com/about/press/white_papers.html, Jan. 2000.

[12] E.R. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye, "Advances and Future Challenges in Binary Translation and Optimization," *Proceedings of the IEEE*, vol. 89, no. 11, Nov. 2001, pp.1710-1722.

[13] C. Iseli and E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis," *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 1995, pp. 173-179.

[14] S. Scalera and J.R. Vásquez, "The Design and Implementation of a Context Switching FPGA," *Proceedings of the 6th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, 1998, pp. 78-85.

[15] *POV-Ray 3.5 Documentation*, Hallam Oaks Pty. Ltd, http://www.povray.org/documentation/, April 2002.

[16] *Red Hat Documentation: Linux Programmer's Manual, PTRACE*, Red Hat, Inc., http://www.europe.redhat.com/documentation/man-pages/man2/ptrace.2.php3, March 2000.

[17] B.F. Veale, J.K. Antonio, and M.P. Tull, "Code Optimization for a Reconfigurable Microprocessor," Technical Report No. CS-TR-03-001, in preparation.