Design and Optimization of Legacy Compatible Microprocessors

Technical Report No. CS-TR-02-002

December 2002

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

Tel: 405-325-8446
Fax: 405-325-4044
E-mail: {veale, antonio, tull}@ou.edu

Table of Contents

Abstract

Microprocessors can be divided into two main categories: (1) those implemented using static hardware, and (2) those implemented using reconfigurable hardware. In microprocessors that use reconfigurable hardware, the instructions supported and the circuitry that performs the instructions can be changed after fabrication. Before a program is run on a microprocessor, it is translated into binary machine code for the microprocessor. There are two approaches to the translation of program code into binary machine code: (1) static and (2) dynamic. In the static translation approach, the program is translated (i.e., compiled) into binary machine code and then the microprocessor executes it directly. In the dynamic translation approach, the microprocessor executes programs that have been initially translated into binary machine code for a different microprocessor, by re-translating the initial binary machine code at execution time.

At the beginning of this report, a taxonomy of the different types of microprocessors (based on these classifications) is presented. The focus of this report is the design and implementation of the IBM DAISY and Transmeta Crusoe microprocessors. Both of these microprocessors use the dynamic translation process to execute programs originally compiled for the PowerPC and Intel X86 microprocessors, respectively. This presentation of the DAISY and Crusoe microprocessors is followed by a comparison of these two microprocessors. Finally, areas for future research are identified and discussed at the end of this report.

List of Figures

List of Tables

1. 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.

The rest of this section presents an overview of a microprocessor taxonomy 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.

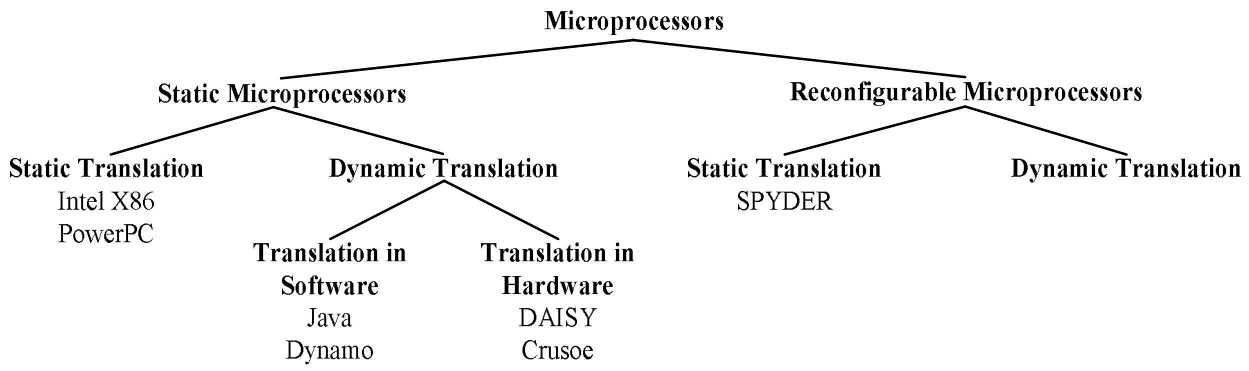


Figure 1. A taxonomy of microprocessors and the translation processes they use.

1.1. 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 binary machine code (commonly referred to as 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.

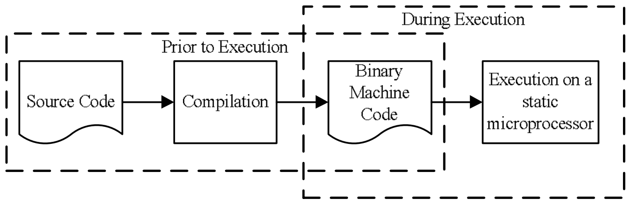


Figure 2. The static translation process for a static microprocessor.

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.

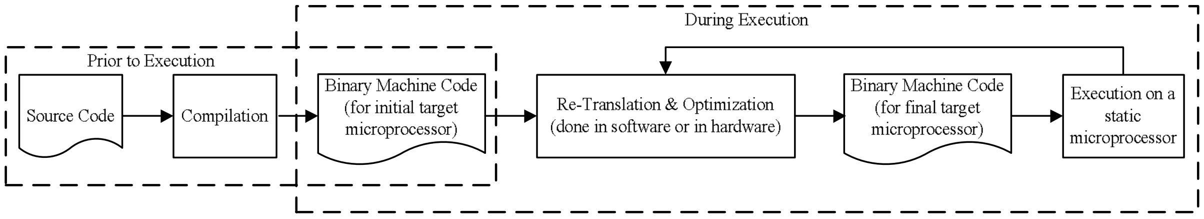

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
Re-Translation & Optimization (done in software or in hardware)
Binary Machine Code (for final target microprocessor)
Execution on a static microprocessor

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

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 of the instructions. The re-translation and optimization step can be performed in software or hardware, as illustrated in Figure 1.

Two examples of systems that perform the re-translation and optimization step in software are JVM [3] and Dynamo [4]. When a Java program is executed on a static microprocessor, the initial machine code, which is called Java byte code, is re-translated into the machine code for the target microprocessor using the JVM, which is implemented in software [5].

In the Dynamo system, the initial and final targeted microprocessors are actually the same. However, when the initially compiled code is executed, the Dynamo software dynamically re-translates and optimizes the initial machine code into machine code with the objective of producing code that executes faster [4].

The DAISY (Dynamically Architected Instruction Set from Yorktown) [6] and Crusoe [7] 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 and then executed by the microprocessor [6, 7]. This re-translation is performed in hardware. A main focus of this report is to overview and compare the DAISY and Crusoe systems (Sections 2 and 3).

1.2. 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.

An example of a reconfigurable microprocessor is the SPYDER (Reconfigurable Processor DEvelopment SYstem) microprocessor [8]. In the SPYDER microprocessor, the circuitry implementing all of the instructions is dynamic. New instructions can be created and the implementation of current instructions can be changed by providing a hardware description for the instructions in the form of binary configuration code that specifies how to configure the reconfigurable hardware [8].

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 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.

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 report, future work is outlined in the direction of examining reconfigurable microprocessor architectures capable of dynamic translation.

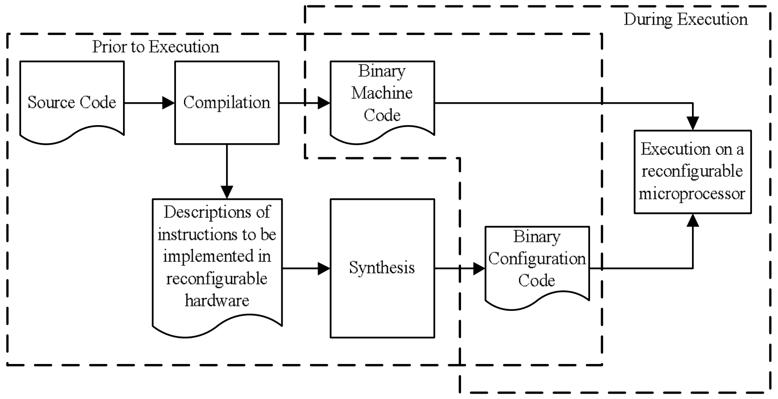

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code
Execution on a reconfigurable microprocessor
Descriptions of instructions to be implemented in reconfigurable hardware
Synthesis
Binary Configuration Code

Figure 4. The static translation process for reconfigurable microprocessors.

1.3. Summary

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.

Microprocessors that perform dynamic translation have the advantage that they can execute machine code that was initially compiled for a different microprocessor. 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 instruction set and the way that instructions are performed. However, current technology that supports reconfigurable microprocessors is slower than the technology used to create static microprocessors. The slower execution of reconfigurable hardware is one reason why reconfigurable technology has not been widely applied to microprocessors in the commercial market.

This report focuses on microprocessors based on the dynamic translation approach to source code compilation. The majority of the material is presented by providing details on the design of the hardware architectures of the DAISY [6] and Crusoe [7] microprocessors. Copies of [6] and [7] can be found in Appendices A and B. At the end of this report, a research idea dealing with the design of instruction sets and machine architectures is discussed. A research idea of how to combine the dynamic translation process with a reconfigurable microprocessor is also presented.

2. The IBM DAISY Microprocessor

2.1. Overview

The DAISY microprocessor [6] 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 be completely compatible with the binary machine code of an existing commercial microprocessor and was the first microprocessor developed exclusively for this purpose [6].

For the purpose of this study, 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 can be applied to a host of different microprocessors such as the Intel X86 and the IBM System/390, as well as virtual microprocessors such as the JVM [6].

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 [6]. 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 VLIW Processor Core.

A microprocessor based on a VLIW processor code (such as the DAISY) packages multiple independent operations into one “very long” instruction for parallel execution in hardware [9]. Each operation is executed using a hardware circuit called an execution unit, also referred to as an Arithmetic Logic Unit (ALU), which can perform several different operations. The operation that is performed at any single point of time by an execution unit is specified using an operation code that is embedded in the VLIW instruction being executed. Such microprocessors use multiple execution units that can independently perform operations, allowing them to perform many operations at the same time. This approach of executing multiple operations in parallel allows for a high degree of ILP (Instruction Level Parallelism).

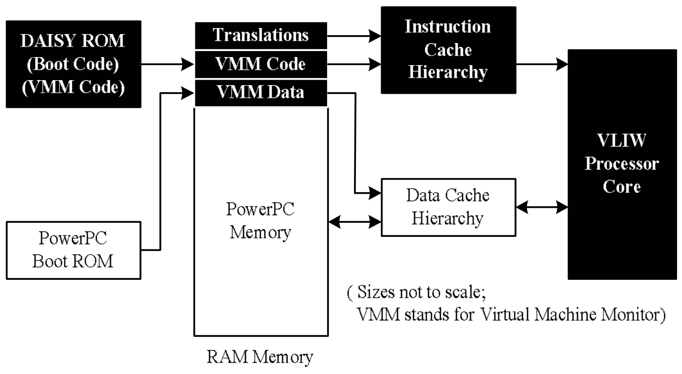

DAISY ROM (Boot Code) (VMM Code)
Translations
VMM Code
VMM Data
Instruction Cache Hierarchy
VLIW Processor Core
Data Cache Hierarchy
PowerPC Memory
PowerPC Boot ROM
(Sizes not to scale; VMM stands for Virtual Machine Monitor)
RAM Memory

Figure 5. The components of a DAISY microprocessor [6].

The execution units in the VLIW processor core used by DAISY are clustered, as shown in Figure 6. Each of the clusters contains four execution units and two load/store units. A cluster is the basic building block within DAISY, and the processor core used in this study has four clusters [6].

The advantages of the clustered design are that: (1) the processor core has a high execution bandwidth and (2) high clock frequency. One disadvantage to this approach is that if an operation dependent on another operation that has been scheduled for a different cluster is encountered, a one-cycle delay occurs in the processing of the dependent operation [6].

In the DAISY microprocessor, instructions are tree-based and implement a multi-way path selection scheme [6]. The flow control model for a tree-based instruction is given in Figure 7. 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 [6]. In the model of Figure 7, each path can consist of any subset of the sixteen operations. The condition codes (ccA, ccB, and ccC) determine the path is taken and what instruction is performed next [10].

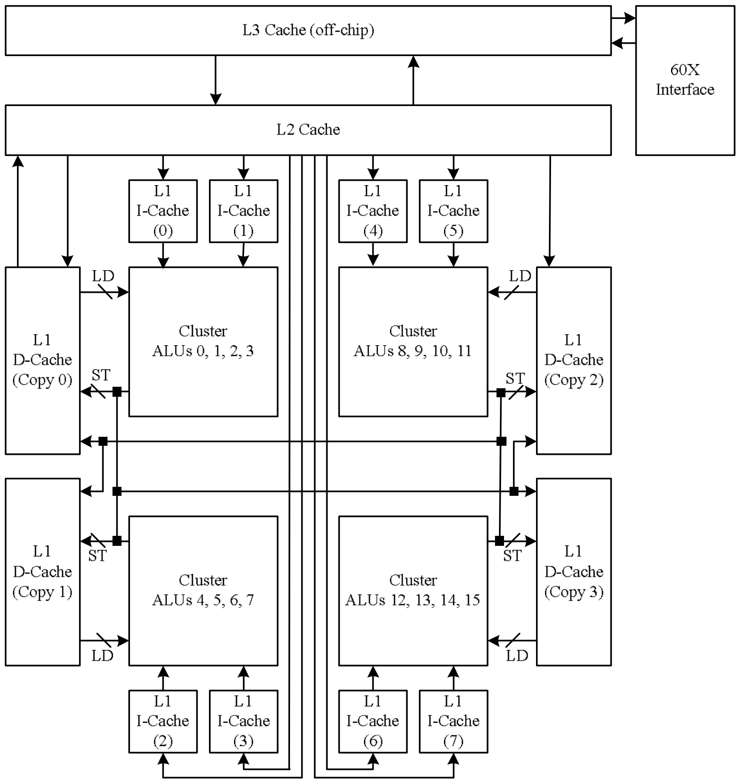
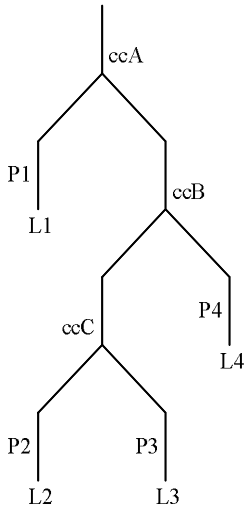


Figure 6. The clustered VLIW processor core of the DAISY microprocessor [10, 6].



```
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 7. The tree-based instruction flow control model and instruction format [10].

The execution process for DAISY VLIW instructions is shown in Figure 8. The process is implemented in hardware, as a pipeline, and is segmented into sets of tasks, called stages. In the first stage, called the instruction fetch (IF) stage, a block of four consecutive instructions is read in from memory and a 4×1 multiplexer chooses the instruction that is to be performed. The next stage, called the execute (EX) stage, combines three tasks: (1) the fetching of the operands from the register file; (2) execution of the sixteen operations; and (3) evaluation of the tree form, which takes place in the branch unit. This stage determines the path of the tree-based instruction that is taken. In the final stage of the pipeline, the write back (WB) stage, the results of the operations on the taken path are written back to the register file [10]. Each stage is performed concurrently, thereby increasing the instruction throughput of the processor core and helping to increase the overall speed of the microprocessor.

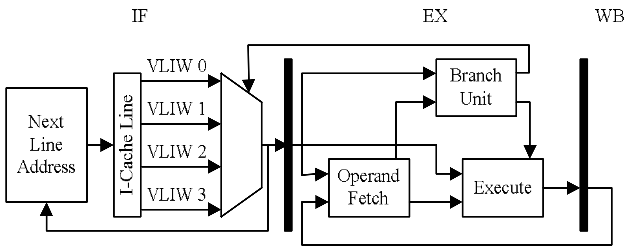


Figure 8. The DAISY Instruction Pipeline [10].

The DAISY microprocessor performs the re-translation and optimization step of Figure 3 by performing a re-translation of machine code (compiled for a different static microprocessor) 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 9 [6].

Overviews of the underlying hardware architecture of the DAISY microprocessor and the dynamic translation process have been presented in this subsection. In the next subsection, a discussion of how the system performs the re-translation of machine code is provided. Subsection 2.3 defines the process of optimization, followed by an overview of special hardware and control mechanisms that are provided in the DAISY system in Subsection 2.4. Finally, a performance evaluation of the DAISY microprocessor is presented in Subsection 2.5.

2.2. Re-translation 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 [6].

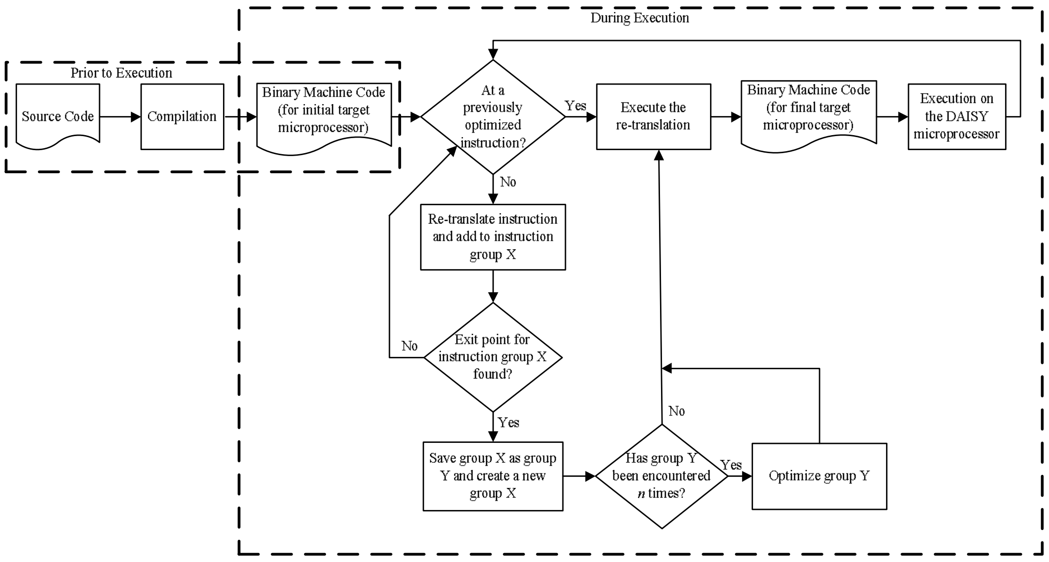
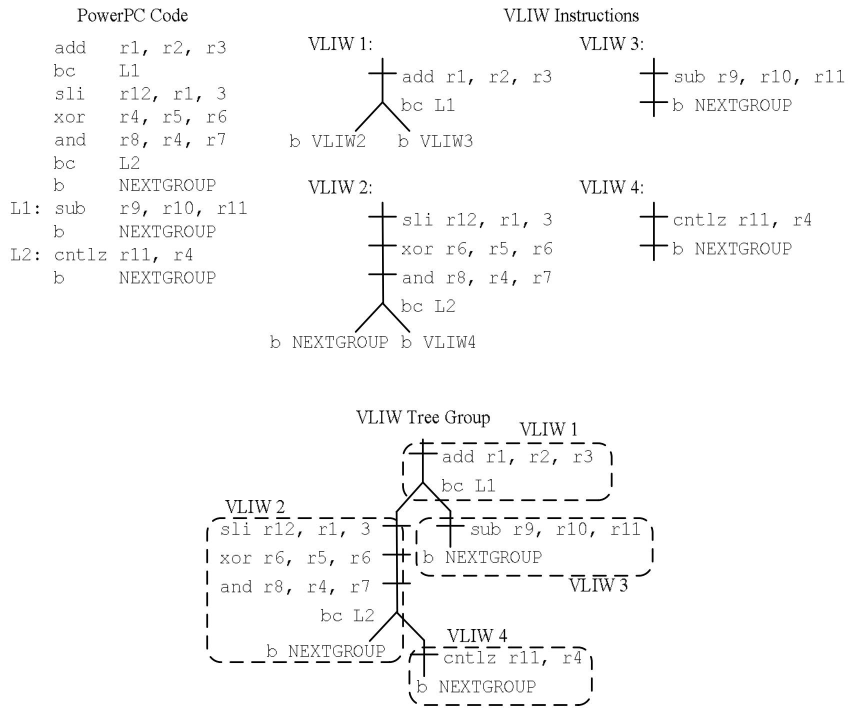

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
At a previously optimized instruction?
Yes
No
Execute the re-translation
Binary Machine Code (for final target microprocessor)
Execution on the DAISY microprocessor
Re-translate instruction 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 9. The dynamic translation process used by the DAISY microprocessor derived from [4, 6].

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 natural 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 [6].

An example segment of PowerPC code and the corresponding PowerPC VLIW instructions and VLIW tree group is shown in Figure 10. In the figure, the PowerPC code is packed into four VLIW instructions. The contents of the VLIW instructions are dependent on where branches occur in this example. Note that a tree group does not necessarily contain only four VLIW instructions. This particular example shows a group of VLIW instructions, which is independent of the four paths within a given VLIW instruction (see Figure 7). The resulting instructions and tree group, shown in the figure, have not been optimized at this point in the execution process.

The first time a segment of machine code is encountered it is re-translated from PowerPC code into machine code for DAISY and is added to an instruction group, as shown in Figure 9. Once a stopping point for the group is found that meets certain requirements, the group is executed by the DAISY microprocessor. The selection of stopping points for tree groups in DAISY is governed by a set of simple principles. A tree group can end at the target of a backward branch (which is usually the beginning of a loop), at a subroutine entry, or at a subroutine exit. Note that subroutine entries and exits can only be determined heuristically by examining the branch, link, and register-indirect branch instructions of the PowerPC code. Additionally, tree groups can span processor pages, protection domains, and indirect branches (which are handled by using runtime information to replace the branch with a set of conditional branches) [6].



```
    add   r1, r2, r3
    bc    L1
    sli   r12, r1, 3
    xor   r4, r5, r6
    and   r8, r4, r7
    bc    L2
    b     NEXTGROUP
L1: sub   r9, r10, r11
    b     NEXTGROUP
L2: cntlz r11, r4
    b     NEXTGROUP
```

```
VLIW 1:
  add r1, r2, r3
  bc L1
  b VLIW2   b VLIW3

VLIW 2:
  sli r12, r1, 3
  xor r6, r5, r6
  and r8, r4, r7
  bc L2
  b NEXTGROUP  b VLIW4

VLIW 3:
  sub r9, r10, r11
  b NEXTGROUP

VLIW 4:
  cntlz r11, r4
  b NEXTGROUP
```

Figure 10. Example PowerPC code and corresponding VLIW instructions and tree group [6].

Finding a stopping point does not mean that the tree group will end at such a point. Either the desired ILP must have been reached for the tree group or the tree group must reach a certain “window size” before the tree group is ended. The term “window size” refers to the number of PowerPC operations found on the path being considered from the root of the tree group. The condition on the ILP is aimed at attaining maximum performance and the condition on the “window size” is meant to limit code explosion. Both of these limits are dynamically adjusted according to the frequency of code execution. This approach also has the benefit of implicitly performing loop unrolling [6].

Tree groups are used as the unit of translation in the re-translation process. This helps to simplify the scheduling of speculative operations because any predecessor instruction dominates all its successors. Additionally, tree groups can have at most one reaching definition [6], meaning that, if a variable is defined at any point within a tree group, then it cannot be re-defined within the same tree group [11]. This helps to simplify optimization (and scheduling) approaches [6].

Originally, processor pages were used as the unit of translation in DAISY. However, tree groups were adopted later because it was discovered that paths through the program that are not frequently executed ended up being re-translated. This re-translation of infrequently executed code led to a large amount of unnecessary code and limited the ILP achieved by the microprocessor [6].

With the advent of tree groups, 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. As before, if re-translated code is executed frequently, then it is optimized [6].

The process of re-translating code a certain number of times before it is optimized is beneficial to the overall performance of the system. 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. After a tree group has been encountered a set number of times, it is optimized [6].

2.3. Optimization of Tree Groups

As shown in Figure 9, once a threshold on the number of times to execute an un-optimized segment of PowerPC code is reached, the associated tree group is optimized. The goal of the optimization algorithms used in DAISY is to attain a significant level of ILP 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 [6].

As each operation is optimized, it is examined in-order (i.e., non-speculatively) and immediately placed into a VLIW instruction. At the same time, DAISY performs global VLIW scheduling on multiple paths and across loop iterations. If the resulting operations are scheduled in-order, then the results will be in the correct destination register after the operation is executed. However, if the operation is scheduled out-of-order (i.e., speculatively), then the result is placed into a hidden register, that can only be seen by the DAISY microprocessor and not the emulated PowerPC microprocessor. It is later copied into the correct destination register associated with the original in-order execution of the program [6].

Tree groups are initially created with moderate ILP and “window size” parameters. If the time spent on a path in a tree group is above a certain threshold, then the tree group will be extended and optimized again using a higher ILP goal and a larger “window size.” This allows the translator to spend more time very aggressively optimizing frequently executed code, while still optimizing less frequent code at a moderate level [6].

The optimization process used by DAISY performs several optimizations. Two of the optimizations performed are copy propagation and load-store telescoping [6]. Copy propagation is a code transformation that first searches for operations following a copy operation that use the destination register of the copy operation as a source register. When such an operation is found, the source register of the operation is replaced with the source register of the original copy operation [11]. In the DAISY system, copy propagation is also used to recognize when instructions that use the same registers do not have any real dependence between them. For example, the PowerPC instructions in the code shown in Figure 11(a) use the same registers, but have no real dependence between them. Because there is actually no dependence between these instructions, they can be performed in parallel in the single VLIW instruction of Figure 11(b) [12].

```
PowerPC Code (before)                 VLIW Instruction (after)

addi  r3, r4, 0         addi r3, r4, 0  xoril r5, r4, 0xFFFF
xoril r5, r3, 0xFFFF

         (a)                                   (b)
```

(The instructions shown here have the format of:
`<operation> <destination register>, <source register>, <source register>`)

Figure 11. Example of Copy Propagation between PowerPC instructions with no real dependencies [12].

Load-store telescoping is an optimization that looks for load operations that correspond to previous store operations. When such patterns are found, the dependency of the instructions involved in the load-store chain can be re-arranged such that no load or store operations need to be performed. Figure 12 provides an example of using load-store telescoping to optimize PowerPC code. Assuming that none of the instructions between `stw` and `lwz` write to `r1` and that no other store instructions are found between these two instructions that write to `8(r1)`, then the code of Figure 12(a) can be re-written as VLIW instructions as shown in Figure 12(b) [12].

```
PowerPC Code (before)                    VLIW Instructions (after)

xor   r3, r5, r6      xor r3, r5, r6
stw   r3, 8(r1)       stw r3, 8(r1)    copy r4', r3    xoril r7', r3, 0xFC
...                   ...
lwz   r4, 8(r1)       copy r4, r4'     copy r7, r7'
xoril r7, r4, 0xFC

        (a)                                       (b)
```

Figure 12. Example of Load-Store Telescoping optimizing PowerPC code [12].

Load-store telescoping has the benefit of being able to remove load and store operations for values maintained in memory that are used every time the values are needed. This removes these operations from the critical path of the program allowing programs optimized with this single technique to approach the performance of fully optimized code [12].

The optimization process also performs re-scheduling of operations in order to increase the ILP of the optimized code. If a speculatively executed operation results in an incorrect execution (i.e., not the original in-order behavior) then an exception is raised and the results are corrected. Each time this happens a counter is incremented and if a tree group has a large number of poorly scheduled speculative instructions, then the entire tree group will be rescheduled conservatively with these speculative operations scheduled in-order [6]. This adaptive approach allows the DAISY system to make mistakes in scheduling due to the aggressiveness of the process and still gracefully recover from and correct such mistakes.

The scheduling approaches also support re-arranging the order of load instructions optimistically and must handle incorrectly scheduled loads appropriately. An exception is raised on a load operation whose target memory location has been altered between the point when the load is executed and when the result is committed. When an exception of this type is caught, the system takes corrective actions to ensure the load occurs correctly [6].

In Figure 13, the re-translated VLIW code for the PowerPC code segment of Figure 10 is shown. This example shows how the `xor` operation can be performed in the first VLIW instruction and the four VLIW instructions of Figure 10 can be compressed into two VLIW instructions. The movement of the `xor` operation shows how the DAISY re-translation process performs operations as early as possible, with the result placed in a re-named register (`r63` in Figure 13) if the operation is moved to an earlier VLIW instruction. Then the results of the moved instruction are placed into the correct PowerPC register (`r4` in Figure 13) at the correct place in the re-translated code as seen with the `r4 = r63` operation. This mechanism allows the microprocessor to perform precise exception handling [6].

Original PowerPC Code

```
    add   r1, r2, r3
    bc    L1
    sli   r12, r1, 3
    xor   r4, r5, r6
    and   r8, r4, r7
    bc    L2
    b     NEXTGROUP
L1: sub   r9, r10, r11
    b     NEXTGROUP
L2: cntlz r11, r4
    b     NEXTGROUP
```

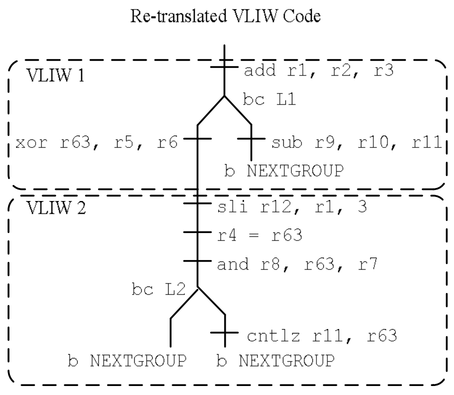


Figure 13. Example PowerPC code and the corresponding translated VLIW Code [6].

As a result of the optimization process, programs that are flat and do not have comparatively highly executed code fragments will not be optimized aggressively. This helps to preserve cache resources and reduce translation overhead [6].

2.4. 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.

An exception is an event that requires special processing that changes the normal flow of execution, e.g., division by zero [13]. An important feature of DAISY is its precise exception handling mechanism.

When an exception is encountered while executing program code, the VMM determines the PowerPC instruction that was being performed when the exception occurred. Next, the actions that would be required by the PowerPC are performed. Finally, the microprocessor branches to the operating system code that handles the exception. However, if the instruction that caused the exception was being speculatively performed, special processing of the exception must be performed [6].

If a speculative operation causes an exception, then the register it writes to is tagged by setting an "exception tag" bit included in the register. This tag bit tells the VMM that the result in the associated register is incorrect. Then, if a non-speculative operation uses a tagged register, an exception is raised and the VMM handles the exception appropriately. This approach allows the optimization process to aggressively schedule instructions without affecting the exception behavior of the initial PowerPC machine code [6].

A context switch occurs when the microprocessor switches from executing one program to another; saving the context of the current program to memory and loading the context of the new program in from memory [13]. The DAISY supports this mechanism by only using non-PowerPC registers as destination registers for speculative operations. As speculative operations write to registers, non-PowerPC registers are used and their values are copied to the correct PowerPC register at the point in time that the operation would have written to the register if the PowerPC code was being executed in-order. This feature combined with the precise exception handling mechanism removes the need to save or restore non-PowerPC registers when a context switch occurs [6]. This means that DAISY does not have to do any special processing when a context switch occurs and such an event is handled solely by the operating system.

Another area in which DAISY provides a specialized control mechanism is in the handling of register-indirect branches. A register-indirect branch is a branch in which the target of the branch is specified in a register. When this type of branch is encountered, the system uses the data in the specified register to determine the target location within the machine code of the branch.

When scheduling a register-indirect branch, the microprocessor does not know the branch target until the branch is executed. This can cause the optimization process to schedule such operations exclusively in-order (such that the branch is the only operation being performed) [9] which significantly impedes performance [6]. To avoid such serializations, the DAISY converts register-indirect branches into a series of conditional branches followed by a register-indirect branch to ensure that the branch occurs correctly if the target is not provided by one of the conditional branches. If additional branch targets are discovered in the future, then the series of conditional branches is updated to test for the additional targets [6].

Self-referential and self-modifying program code can cause problems in emulated microprocessors because the code is re-translated from one binary machine format to another and the self-referential or self-modifying code is not aware of the changes made. Examples of self-referential code include code that performs a checksum on itself, has constants intermixed within it, and relative branches. The handling of such code in the DAISY microprocessor is straightforward because the PowerPC code can only refer to itself through the PowerPC registers; and in DAISY these registers contain the values they would if the program code was running on the microprocessor for which it was initially compiled [6].

The handling of self-modifying code is more complicated than self-referential code. The DAISY microprocessor handles this situation through the use of a "read-only" bit included in every unit of memory allocated to the PowerPC microprocessor. This "read-only" bit is hidden from the PowerPC microprocessor being emulated and tells the VMM when the tree group(s) associated with the unit of memory should be invalidated [6].

When machine code in memory is re-translated, the "read-only" bit for the unit of memory holding the code is set. Then, if a store operation occurs within a unit of memory whose "read-only" bit is set as the destination of the store, the store is committed and the execution of the re-translated code is interrupted. Next, the VMM invalidates the tree group(s) associated with the modified memory (the destination memory of the store). Finally, the PowerPC code resumes execution with the instruction immediately following the store instruction, resuming the re-translation-optimization-execution cycle; and when the modified code is to be executed in the future, it will be re-translated again with the modifications in place [6].

With special support for exception handling and context switching, indirect branches, self-modifying, and self-referential code, the DAISY microprocessor can overcome potential problems that would otherwise degrade performance. Without the mechanisms provided for such situations, the DAISY microprocessor would most likely perform at an unacceptable level.

2.5. Performance Evaluation

This subsection presents some of the aspects of the DAISY microprocessor that affect performance of the binary machine code running on the microprocessor. The aspects of DAISY studied include the re-translation of machine code, the optimization process, and the underlying hardware architecture.

In the studies performed in [6], re-translation of code was used to filter out portions of code that are not frequently executed from the translation cache (this process was not used to gather program profiling information to help guide tree group formation and optimization in these studies). However, filtering of infrequently executed code was not found to result in better cache performance, as might be expected. The result of filtering is larger segments of machine code for the regions of the initial machine code that were ultimately re-translated. This increase in code segment sizes make the performance of the instruction cache a more important factor in the performance of the system than when filtering of code is not performed negating the savings in time from filtering out infrequently executed code [6].

The studies also discovered that where the tree groups are terminated has an effect on the dynamic path length of the tree groups. The dynamic path length is the average number of PowerPC instructions between the root and leaves of a tree group. This measure is important because longer paths give the translator more opportunity to speculatively schedule VLIW instructions [6] and increase the ILP achieved. However, these speculative operations are only useful if they lie on the path taken at execution time. Additionally, incorrectly predicted speculative instructions can reduce the dynamic path length. This can occur when the number of paths through the program code exceeds what can be covered by tree groups [6]. Thus, the selection of good stopping points for tree groups is directly correlated with performance of the system.

Several different configurations of the VLIW processor core were studied for the DAISY project. The different configurations considered are listed in Table 1 and range from 4-issue processor cores to 16-issue processor cores. All of the execution units have support for arithmetic and logic operations, and one or two units per cluster can perform memory operations [6].

As might be expected, the wider configurations of the VLIW processor core were found to provide a significant improvement over competing microprocessors in terms of high clock frequency. However, the interesting result is that the narrower configurations also performed well. This result is due to the lower translation overhead of narrower configurations. Because of the lower translation overhead, these configurations have a very good CPI (CPI stands for clock

Cycles Per Instruction and is a measure of the average time needed to perform an instruction [9]) compared to current superscalar microprocessors [6]. (Superscalar microprocessors execute a varying numbers of instructions at the same time that are statically scheduled at compile time or dynamically scheduled by the microprocessor at execution time, while VLIW-based microprocessors attempt to execute a fixed number of instructions at the same time that are typically statically scheduled at compile time [9].) Additionally, the simplistic hardware of the narrower configurations, if implemented in silicon, should result in a higher frequency microprocessor than the implementation of the wider configurations [6].

Table 1. VLIW processor core configurations explored for the DAISY microprocessor [6].

	Configurations					
Number of Clusters	1	1	2	2	4	4
Number of ALUs/Cluster	4	4	4	4	4	4
Number of L/S Units/Cluster	1	2	1	2	1	2
Number of Branch Units	1	1	2	2	3	3
I-Cache Size	8K	8K	16K	16K	32K	32K

The performance studies of DAISY, presented in [6], 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. Experiments simulating different configurations of the DAISY microprocessor have also shown that both the wide and narrow configurations, of Table 1, perform well. These studies have shown that most of the approaches used in the DAISY microprocessor are useful while the filtering of machine code may not always be beneficial to the overall performance of the system.

2.6. Summary

The DAISY microprocessor uses the dynamic translation process of Figure 3 solely for the purpose of executing binary machine code compiled for a different microprocessor [6]. As a result of the approach taken to dynamic translation in the DAISY microprocessor, the processes used to re-translate and execute machine code are transparent to the machine code of the initial microprocessor and the resulting microprocessor is completely compatible with the initial microprocessor.

The keys to the success of the approaches used in DAISY are that the system performs ILP extraction at execution time [6] 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 VLIW compilers use, that result in trade-offs being considered to improve performance [6].

The DAISY project is innovative in its combination of a clustered VLIW processor core, tree-based VLIW instructions, tree groups as a unit of translation, and its scheduling and exception handling mechanisms. This work represents a new direction in which legacy-compatible microprocessor design may go in the future. In fact, Transmeta Corporation has already taken this general approach in producing a commercial line of Intel X86 compatible microprocessors [7] and is the topic of the next section.

3. The Transmeta Crusoe Microprocessor

3.1. Overview

The Crusoe microprocessor [7], 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 [7].

A high-level view of a Crusoe based system is shown in Figure 14. 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 parts: (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; (4) and the Morph host which is the VLIW processor core of the microprocessor [7].

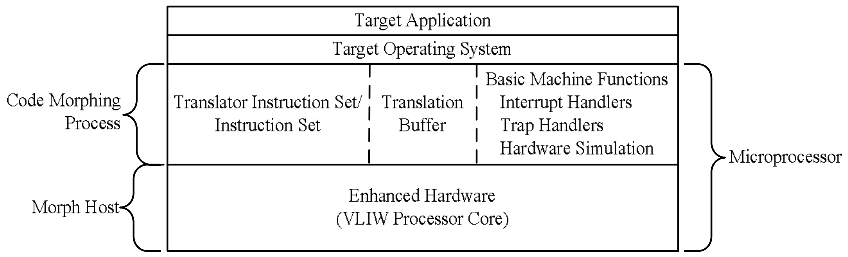


Figure 14. The components of a Crusoe based system [7].

The registers in the Crusoe consist of the same registers as the Intel X86, called official registers, and a set of working registers, some of which duplicate (or shadow) the official registers, as seen in Figure 15. As the Crusoe performs operations, it uses the working registers and preserves the previous state (i.e., the official state) of the emulated X86 microprocessor in the official registers. When a code segment boundary (e.g., a subroutine entry or exit) in the X86 machine code is encountered, the official state of the emulated X86 microprocessor is updated by copying the values of the working registers to the official registers. This mechanism is supported by an extra stage in the instruction pipeline of the microprocessor to avoid slowing down operation of the microprocessor [7]. This approach to execution of re-translated machine code is different from the DAISY in that the Crusoe does not work directly on the registers containing the official state of the microprocessor (making rollback of operations performed on registers trivial) and the DAISY only uses extra registers for the speculative execution of operations.

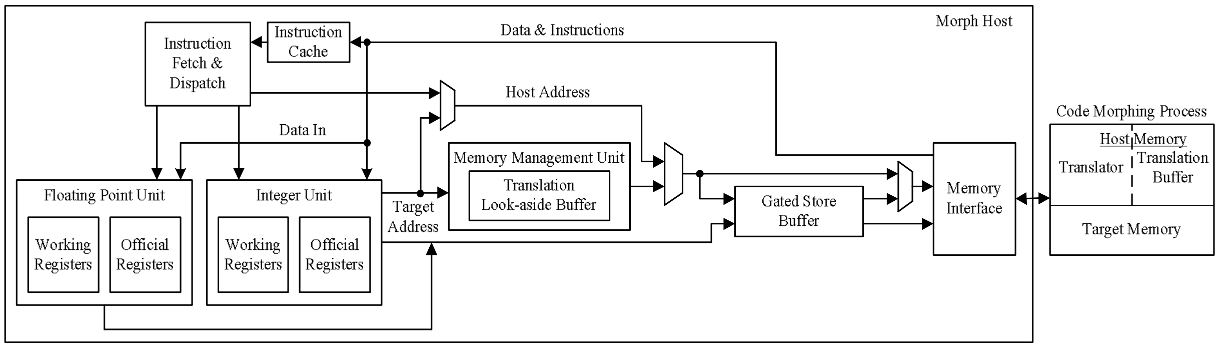


Figure 15. Architecture of the Crusoe microprocessor [7].

Another important component of the Crusoe is the gated store buffer, shown in Figure 15, which buffers writes to memory by holding the address and data for each store to memory. This queue of memory stores temporarily holds memory state changes before they are committed to the official memory state of the emulated X86 microprocessor, as illustrated in Figure 16. This mechanism ensures that the state of the emulated microprocessor is correct at the time of interrupts and exceptions. The stores between the head of the queue and the gate pointer have already been committed to memory and those between the gate pointer and the tail of the queue are those that have not been committed [7].

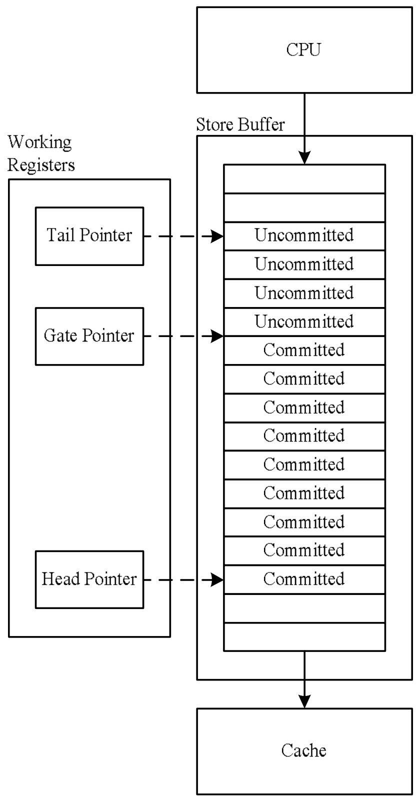


Figure 16. The gated store buffer used to buffer writes to memory and its associated registers [7].

Commit operations occur on code segment boundaries. When a commit operation occurs, the uncommitted stores are committed to memory and the gate pointer is moved to the tail of the queue. If a rollback operation is needed, e.g., for processing an exception, then the uncommitted stores are removed from the queue and the tail pointer is moved to the position of the gate pointer [7]. In contrast, the DAISY microprocessor does not provide a mechanism similar to the gated store buffer and must manually rollback any writes to memory whereas the Crusoe only has to change the value contained in one register.

The Code Morphing process of the Crusoe microprocessor maintains a translation buffer, as shown in Figure 15, 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 [7].

The use of a translation buffer approach greatly improves the speed of the microprocessor because it does not have to fetch, decode, re-translate, optimize, re-order, and schedule operations every time they are executed [7]. This mechanism is similar to and serves the same purpose as the instruction cache found in the DAISY. The structure used for the translation buffer may be implemented in hardware (e.g., as an instruction cache) or in software (e.g., as a data structure residing in memory).

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. This process is the re-translation step of Figure 3 and is shown in Figure 17 for the Crusoe [7].

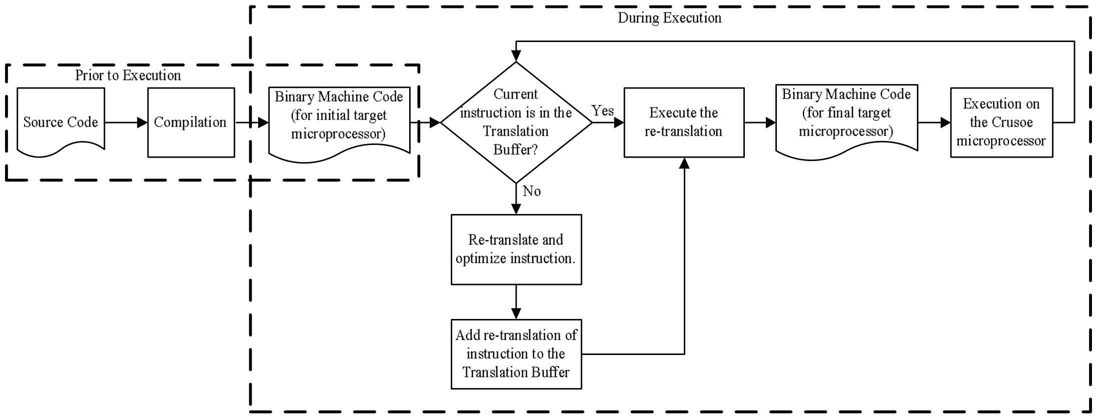

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
Current instruction is in the Translation Buffer?
Yes
No
Execute the re-translation
Binary Machine Code (for final target microprocessor)
Execution on the Crusoe microprocessor
Re-translate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

Figure 17. The dynamic translation process used by the Crusoe microprocessor derived from [7].

The performance of the Crusoe microprocessor comes from the reduction in the amount hardware in the microprocessor (compared to the Intel X86 microprocessor) and the caching of re-translated machine code. This results in a possible speed up of the execution of program code and a reduction in the power consumption of the microprocessor [14].

Transmeta has not made conclusive performance benchmark results for the Crusoe microprocessor readily available. Instead of proclaiming a faster execution time for applications and operating systems, Transmeta focuses on proclaiming the lower power consumption rate of the Crusoe microprocessor compared to state of the art compatible microprocessors. According to Transmeta, the Crusoe microprocessor consumes 60%-70% less power than other conventional microprocessors. Due to this reduction in power consumption, Transmeta focuses their efforts on the lightweight mobile computer and handheld markets [14].

An overview of the dynamic translation process and the underlying hardware architecture of the Crusoe microprocessor have been presented in this subsection. In the next two subsections, a discussion of how the system performs re-translation (Subsection 3.2) and optimization (Subsection 3.3) is provided. In Subsection 3.4, special control mechanisms and support is presented, followed by a discussion of exception handling in Subsection 3.5.

3.2. Re-Translation of Instructions

The Crusoe microprocessor uses the Code Morphing process, shown in Figure 14, 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 [7]. 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 17. 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 [7].

An example C program and the corresponding Intel X86 assembly code are shown in Figure 18. As the code is executed by the Crusoe microprocessor, each X86 instruction is re-translated into a series of Crusoe operations. These operations perform the X86 segmentation process, memory bound checking, and the operations required to create the results specified by the X86 instruction [7]. Figure 19 shows each X86 instruction followed by the necessary Crusoe operations.

Unlike DAISY, the Crusoe does not use tree groups as the unit of translation, but instead uses X86 instructions. Additionally, although the instruction format used is by the Crusoe is not specified in [7], there is no indication that the Crusoe uses an instruction format similar to the tree-based instructions of DAISY.

```
          Original C Code                              Corresponding Assembly-Level Code
while( (n--)[001b]>0) {     mov %ecx, [%ebp+0xc] // load c from memory address into %ecx
   *s++=c                   mov %eax, [%ebp+0x8] // load s from memory address into %eax
}                           mov [%eax], %ecx     // store c into memory address held in %eax
                            add %eax, #4         // increment s by 4
                            mov [%ebp+0x8], %eax // store (s + 4) back to memory
                            mov %eax, [%ebp+0x10]// load n from memory address into %ecx
                            lea %ecx, [%eax-1]   // decrement n and store result in %ecx
                            mov [ebp+0x10], %ecx // store (n - 1) into memory
                            and %eax, %eax       // test n to set condition codes
                            jg  .-0x1b           // branch to top of this section if "n > 0"
```

Figure 18. Example C program and corresponding assembly-level code [7].

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. The optimization process is presented in the next subsection.

3.3. Optimization of Crusoe Instructions

The Code Morphing process not only re-translates X86 instructions into Crusoe operations, it also optimizes the operations using several techniques including common sub-expression elimination, speculative removal of commit operations, and copy elimination. Such 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.

Although [7] does not specify in detail how the Crusoe microprocessor determines which re-translations should be optimized, the following three criteria are discussed in [7]. First, a count of how many times a re-translation is executed is kept. If this count reaches some threshold, then an exception can be raised and the re-translation can be optimized at that time. This mechanism can be embedded into the re-translations as software. Second, the Code Morphing process can interrupt the execution of re-translations at a specified frequency and optimize the re-translation running at the time the system is interrupted if it has not already been optimized. Finally, the Code Morphing process can simply optimize certain types of operations or sequences of operations (e.g., loops) [7].

Re-translated X86 Code

```
mov   %ecx, [%ebp+0xC]       // load c from memory address into %ecx
add   R0, Rebp, 0xC          ; form logical address into R0
chkl  R0, Rss_limit          ; check logical address against segment lower limit
chku  R0, R_FFFFFFF          ; check logical address against segment upper limit
add   R1, R0, Rss_base       ; add segment base to form linear address
ld    Recx, [R1]             ; load c from memory address in R1 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x8]       // load s from memory address into %eax
add   R2, Rebp, 0xB          ; form logical address into R2
chkl  R2, Rss_limit          ; check logical address against segment lower limit
chku  R2, R_FFFFFFF          ; check logical address against segment upper limit
add   R3, R2, Rss_base       ; add segment base to form linear address
ld    Reax, [R3]             ; load s from memory address in R3 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%eax], %ecx           // store c into memory address held in %eax
chku  Reax, Rds_limit        ; check logical address against segment upper limit
add   R4, Reax, Rds_base     ; add segment base to form linear address
st    (R4), Recx             ; store c into memory address s
add   Reip, Reip, 2          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
add   %eax, #4               // increment s by 4
addcc Reax, Reax, 4          ; increment s by 4
add   Reip, Reip, 5          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%ebp+0x8], %eax       // store (s + 4) back to memory
add   R5, Rebp, 0x8          ; form logical address into R5
chkl  R5, Rss_limit          ; check logical address against segment lower limit
chku  R5, R_FFFFFFFF         ; check logical address against segment upper limit
add   R6, R5, Rss_base       ; add segment base to form linear address
st    [R6], Reax             ; store (s + 4) to memory address in R6
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x10]      // load n from memory address into %ecx
add   R7, Rebp, 0x10         ; form logical address into R7
chkl  R7, Rss_limit          ; check logical address against segment lower limit
chku  R7, R_FFFFFFF          ; check logical address against segment upper limit
add   R8, R7, Rss_base       ; add segment base to form linear address
ld    Reax, [R8]             ; load n from memory address in R8 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
lea   %ecx, [%eax-1]         // decrement n and store result in %ecx
sub Recx, Reax, 1            ; decrement n
add Reip, Reip, 3            ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [ebp+0x10], %ecx       // store (n - 1) into memory
add   R9, Rebp, 0x10         ; form logical address into R9
chkl  R9, Rss_limit          ; check logical address against segment lower limit
chku  R9, R_FFFFFFFF         ; check logical address against segment upper limit
add   R10, R9, Rss_base      ; add segment base to form linear address
st    [R10], Recx            ; store (n - 1) in Recx into memory using address in R10
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
and   %eax, %eax             // test n to set condition codes
andcc R11, Reax, Reax        ; test n to set condition codes
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
jg    .-0x1B                 // branch to top of this section if "n > "
add   Rseq, Reip, Length(jg) ;
ldc   Rtarg, EIP(target)     ;
selcc Reip, Rseq, Rtarg      ;
commit                       ; commits working state to official state
jg    mainloop, mainloop     ; jump to main loop
```

Figure 19. Example re-translated X86 code (in bold) with Crusoe operations required for each X86 instruction [7].

The rest of this subsection presents each of the different optimizations that are specifically addressed in [7]. These optimizations 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 [7]. Note that the Crusoe optimization process may perform more optimization on code than what is described in [7].

3.3.1. Removing the X86 Segmentation Process

In a segmented memory model, the memory allocated to an application is segmented into a group of independent sections of memory, called segments. In the segmented memory model used by the X86 microprocessor, the code, data, and stacks are assigned to separate segments. In contrast, in a flat memory model the memory is presented to the application as a single section of memory, called a linear address space, in which all of the code, data, and stacks are contained. The segmentation of memory increases the reliability of applications because it prevents the program from overwriting memory that has not been allocated to the program. When a program using the segmented memory model accesses memory, the microprocessor must translate the address appropriately and ensure the correct segment is present in main memory. This process of translation and the mechanisms required to maintain the segments are transparent to an application running on an X86 microprocessor [1].

The Crusoe microprocessor does not include support for transparently translating logical addresses into addresses for a segmented address space or the mechanisms to transparently maintain a segmented memory space. Therefore, the Crusoe inserts appropriate operations to perform these tasks as X86 machine code is initially re-translated. This optimization removes these operations by performing a speculative mapping of all segments of the code segment to the same address space. Note that the speculative removal of these operations requires the assumption that the program currently being optimized is written for a flat memory model, removing the need for the segmentation process.

The example code in Figure 19 (which is the re-translated code resulting from the re-translation process being performed on the X86 code of Figure 18) includes the operations necessary to support a segmented memory model. Assuming this code uses a flat memory model, the optimization process removes these extra operations resulting in the code segment shown in Figure 20 [7]. As seen in these figures, the removal of these operations removes up to three Crusoe operations per X86 instruction.

Optimized X86 Code

```
mov   %ecx, [%ebp+0xC]       // load c from memory address into %ecx
add   R0, Rebp, 0xC          ; form logical address into R0
chku  R0, R_FFFFFFF          ; check logical address against segment upper limit
ld    Recx, [R0]             ; load c from memory address in R1 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x8]       // load s from memory address into %eax
add   R2, Rebp, 0xB          ; form logical address into R2
chku  R2, R_FFFFFFF          ; check logical address against segment upper limit
ld    Reax, [R2]             ; load s from memory address in R3 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%eax], %ecx           // store c into memory address held in %eax
chku  Reax, R_FFFFFFF        ; check logical address against segment upper limit
st    (R4), Recx             ; store c into memory address s
add   Reip, Reip, 2          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
add   %eax, #4               // increment s by 4
addcc Reax, Reax, 4          ; increment s by 4
add   Reip, Reip, 5          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%ebp+0x8], %eax       // store (s + 4) back to memory
add   R5, Rebp, 0x8          ; form logical address into R5
chku  R5, R_FFFFFFFF         ; check logical address against segment upper limit
st    [R6], Reax             ; store (s + 4) to memory address in R6
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x10]      // load n from memory address into %ecx
add   R7, Rebp, 0x10         ; form logical address into R7
chku  R7, R_FFFFFFF          ; check logical address against segment upper limit
ld    Reax, [R7]             ; load n from memory address in R8 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
lea   %ecx, [%eax-1]         // decrement n and store result in %ecx
sub Recx, Reax, 1            ; decrement n
add Reip, Reip, 3            ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [ebp+0x10], %ecx       // store (n - 1) into memory
add   R9, Rebp, 0x10         ; form logical address into R9
chku  R9, R_FFFFFFFF         ; check logical address against segment upper limit
st    [R9], Recx             ; store (n - 1) in Recx into memory using address in R10
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
and   %eax, %eax             // test n to set condition codes
andcc R11, Reax, Reax        ; test n to set condition codes
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
jg    .-0x1B                 // branch to top of this section if "n > "
add   Rseq, Reip, Length(jg) ;
ldc   Rtarg, EIP(target)     ;
selcc Reip, Rseq, Rtarg      ;
commit                       ; commits working state to official state
jg    mainloop, mainloop     ; jump to main loop
```

Figure 20. Example re-translated X86 code (in bold) with the Crusoe operations required for each instruction after removal of the X86 segmentation process [7].

If the speculation that the application being optimized uses a flat memory model is wrong, then the execution of the optimized code will fail. When this happens, the Code Morphing process will rollback the state of the emulated microprocessor and the operations that support the segmented memory model (that were removed) are re-inserted into the code segment. Finally, the code will be re-executed with the newly inserted code in place [7].

3.3.2. Removing Upper Boundary Memory Checks

Pages are virtual units of memory that allow the memory space of the microprocessor to be extended beyond that provided by the semiconductor-based memory devices used to support the main memory. When a page is needed, it is loaded into memory from a secondary memory device (e.g., a hard drive) and when it is no longer needed, it is stored back out to secondary storage [9]. It may be the case that a data unit does not completely fit in one page and is split among multiple pages. If such a splitting occurs, then the data is said to be unaligned; and when the associated data is referenced, all corresponding pages must be present in main memory.

The operations included in the re-translated X86 code include an operation that checks the logical address created by the code against the upper boundary of the address space for the memory segment being used. These memory checks are not removed as part of the optimization that performs the removal of the X86 segmentation process because the memory reference may refer to unaligned memory, in which case the microprocessor must ensure that the correct virtual memory pages are present in main memory [7].

Under the assumption that the instructions and data of the application being optimized are correctly aligned and assuming the application uses a flat memory model, the operations that perform the upper memory boundary checks (the operation `chku`) can be removed [7]. These operations are shown in the example code of Figure 20, which is the re-translated code corresponding to the X86 code of Figure 18 after the removal of the segmentation process. The further optimized code without the upper boundary memory checks is shown in Figure 21. This optimization removes one Crusoe operation per Intel X86 instruction.

If the speculation that the instruction and data of the application are correctly aligned is wrong (or if the assumption that the application uses a flat memory model is wrong), then the execution of the optimized code will fail. When this happens, the state of the emulated X86 microprocessor will be rolled-back and the operations removed during the optimization process will be re-inserted into the code. Finally, the code will be re-executed with the upper boundary memory checks included in the code [7].

Optimized X86 Code

```
mov   %ecx, [%ebp+0xC]       // load c from memory address into %ecx
add   R0, Rebp, 0xC          ; form logical address into R0
ld    Recx, [R0]             ; load c from memory address in R1 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x8]       // load s from memory address into %eax
add   R2, Rebp, 0xB          ; form logical address into R2
ld    Reax, [R2]             ; load s from memory address in R3 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%eax], %ecx           // store c into memory address held in %eax
st    [Reax], Recx           ; store c into memory address s
add   Reip, Reip, 2          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
add   %eax, #4               // increment s by 4
addcc Reax, Reax, 4          ; increment s by 4
add   Reip, Reip, 5          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [%ebp+0x8], %eax       // store (s + 4) back to memory
add   R5, Rebp, 0x8          ; form logical address into R5
st    [R6], Reax             ; store (s + 4) to memory address in R6
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   %eax, [%ebp+0x10]      // load n from memory address into %ecx
add   R7, Rebp, 0x10         ; form logical address into R7
ld    Reax, [R7]             ; load n from memory address in R8 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
lea   %ecx, [%eax-1]         // decrement n and store result in %ecx
sub Recx, Reax, 1            ; decrement n
add Reip, Reip, 3            ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
mov   [ebp+0x10], %ecx       // store (n - 1) into memory
add   R9, Rebp, 0x10         ; form logical address into R9
st    [R9], Recx             ; store (n - 1) in Recx into memory using address in R10
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
and   %eax, %eax             // test n to set condition codes
andcc R11, Reax, Reax        ; test n to set condition codes
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
commit                       ; commits working state to official state
jg    .-0x1B                 // branch to top of this section if “n > ”
add   Rseq, Reip, Length(jg) ;
ldc   Rtarg, EIP(target)     ;
selcc Reip, Rseq, Rtarg      ;
commit                       ; commits working state to official state
jg    mainloop, mainloop     ; jump to main loop
```

Figure 21. Example re-translated X86 code (in bold) with the Crusoe operations required for each instruction after removal of upper boundary memory checks [7].

3.3.3. Common Sub-Expression Elimination

The third optimization presented is common sub-expression elimination [7]. This optimization reduces the number of operations in a sequence by removing unnecessary re-computation of the same expressions [11]. This removal of common sub-expressions is performed on the re-translated machine code in order to further optimize the Crusoe machine code. Common sub-expression elimination does not require any assumptions about the nature of the application being re-translated and the resulting re-translation will always execute correctly [7].

3.3.4. Removing Commit Operations

The fourth optimization used by the Crusoe microprocessor is the speculative removal of commit operations from the re-translated Crusoe machine code. This optimization assumes that the associated machine code segment will not cause an exception. This allows the removal of the commit operations that update the state of the official registers and move uncommitted memory stores, in the gated store buffer, to memory [7].

The removal of commit operations is possible because the state of the emulated X86 microprocessor only needs to be correct when the operating system accesses the state of microprocessor due to the occurrence of an exception. Thus, the commit operation only needs to occur at the end of a sequence of X86 instructions instead of after each instruction. This removes one Crusoe operation for every re-translated X86 instruction and replaces them with only a single commit operation at the end of the sequence of re-translated X86 instructions [7]. These commit operations (`commit`) are shown in the code listing of Figure 21 and the resulting optimized code without these commit operations is shown in Figure 22.

When an exception does occur, the microprocessor will invalidate the uncommitted state of the emulated microprocessor and re-translate the machine code executed since the last commit. In the new re-translation, a commit operation is performed after every sequence of Crusoe operations corresponding to each X86 instruction. Then, when the exception is encountered, the state of the microprocessor is correct and the exception can be properly handled [7].

3.3.5. Register Renaming

The fifth optimization used by the Crusoe microprocessor is the process of register renaming [7]. This optimization relies upon name dependencies between operations that occur when multiple operations utilize the same register(s) and the operations are not dependent on the data computed by the other operations involved [9]. When this occurs, the registers used by such operations can be renamed, removing any hardware dependencies between the operations and allowing them to be executed in parallel utilizing different execution units [7, 9]. This optimization can have a significant impact on the level of ILP depending on how many registers are available for register renaming [9]. DAISY also performs the same type of optimization by using copy propagation to detect name dependencies and register renaming to remove the dependencies so that operations can be performed in parallel.

3.3.6. Code Motion

Code motion is the sixth optimization used by the Crusoe microprocessor [7]. This is a loop optimization that targets expressions (operations) found in the body of the loop that yield the same result in each iteration of the loop. By moving such expressions outside the loop body, the number of operations executed to complete the loop is reduced [11].

Optimized X86 Code

```
mov   %ecx, [%ebp+0xC]       // load c from memory address into %ecx
add   R0, Rebp, 0xC          ; form logical address into R0
ld    Recx, [R0]             ; load c from memory address in R1 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
mov   %eax, [%ebp+0x8]       // load s from memory address into %eax
add   R2, Rebp, 0xB          ; form logical address into R2
ld    Reax, [R2]             ; load s from memory address in R3 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
mov   [%eax], %ecx           // store c into memory address held in %eax
st    [Reax], Recx           ; store c into memory address s
add   Reip, Reip, 2          ; add X86 instruction length to eip in Reip
add   %eax, #4               // increment s by 4
addcc Reax, Reax, 4          ; increment s by 4
add   Reip, Reip, 5          ; add X86 instruction length to eip in Reip
mov   [%ebp+0x8], %eax       // store (s + 4) back to memory
add   R5, Rebp, 0x8          ; form logical address into R5
st    [R6], Reax             ; store (s + 4) to memory address in R6
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
mov   %eax, [%ebp+0x10]      // load n from memory address into %ecx
add   R7, Rebp, 0x10         ; form logical address into R7
ld    Reax, [R7]             ; load n from memory address in R8 into Reax
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
lea   %ecx, [%eax-1]         // decrement n and store result in %ecx
sub Recx, Reax, 1            ; decrement n
add Reip, Reip, 3            ; add X86 instruction length to eip in Reip
mov   [ebp+0x10], %ecx       // store (n - 1) into memory
add   R9, Rebp, 0x10         ; form logical address into R9
st    [R9], Recx             ; store (n - 1) in Recx into memory using address in R10
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
and   %eax, %eax             // test n to set condition codes
andcc R11, Reax, Reax        ; test n to set condition codes
add   Reip, Reip, 3          ; add X86 instruction length to eip in Reip
jg    .-0x1B                 // branch to top of this section if "n > "
add   Rseq, Reip, Length(jg) ;
ldc   Rtarg, EIP(target)     ;
selcc Reip, Rseq, Rtarg      ;
commit                       ; commits working state to official state
jg    mainloop, mainloop     ; jump to main loop
```

Figure 22. Example re-translated X86 code (in bold) with the Crusoe operations required for each instruction after removal of all commit operations except the one at the end of the code segment [7].

3.3.7. Data Aliasing

The seventh optimization used by the Crusoe microprocessor is data aliasing. This optimization recognizes when several operations access the same locations in memory. When such a situation occurs, the Code Morphing process will load the values at the referenced addresses into a register. Then only a register-to-register copy needs to be performed for each operation that references any of the associated memory addresses. The copies of data from memory, residing in the registers, are marked as aliased, and when a change is detected, an exception will occur. This optimization is beneficial because every load operation that involves moving data from the affected memory addresses is changed to a simple register-to-register copy that executes much faster than a load from memory [7].

3.3.8. Copy Propagation

The eighth optimization performed is copy propagation. For copy propagation, the Code Morphing process removes unnecessary register-to-register copies by using the register in which the data originally existed whenever copies of that register are used. (Note that once this has been done, the original copy operation is removed if it is no longer needed using an optimization called dead code elimination [11].) This effectively reduces the number of cycles required to execute a segment of code [7].

3.3.9. Using Alias Hardware

The final optimization, considered in [7], is the use of alias hardware to remove store operations from loops. This optimization is similar to that of data aliasing, except here the aliased registers are used to hold data to be stored into memory. This allows the store operations within the loop body to be replaced with register-to-register copies or register references instead of memory stores. Additionally, the actual stores to memory are moved such that they occur immediately after the end of the loop. This optimization speeds up the processing of loops by either replacing memory stores with register transfers or eliminating the store operations from the body of the loop altogether [7]. The optimization is similar to the load-store telescoping optimization used by the DAISY microprocessor [12].

3.4. Special Hardware and Control Mechanisms

The Crusoe microprocessor includes special hardware and control mechanisms for several different types of X86 program support. These areas include support for rollback operations, memory mapped I/O (Input/Output), and self-modifying program code. The Crusoe supports rollback operations through the use of shadow registers and the gated store buffer, both presented in Subsection 3.1. The present subsection presents the support that the Crusoe provides for memory mapped I/O and self-modifying code.

Memory mapped I/O refers to the way that memory mapped I/O devices are attached to the microprocessor. An I/O device can be mapped to a memory address so that the programmer can access it using typical memory stores and writes [9]. Due to the nature of memory mapped I/O, it is impossible to distinguish memory instructions from memory mapped I/O instructions [7].

Because memory mapped I/O operations often must be performed in the precise order specified in the X86 machine code, a system normally must treat all memory operations with conservative assumptions so as minimize the affect memory on mapped I/O. In order to allow optimization for true memory operations, an A/N (Abnormal/Normal) protection bit is included in every address translation in the Translation Look-aside Buffer (TLB). This protection bit specifies if the memory accesses used for the associated memory address is abnormal (i.e., an access to memory mapped I/O) or normal (i.e., an access to regular memory). This mechanism allows true memory accesses to be speculatively re-translated [7].

When an access to memory is re-translated, the Code Morphing process initially treats it as a regular access to memory, which can be speculatively scheduled. Then after the re-translation executes, the memory access type of the operation is compared against the A/N bit. If the access type used in the re-translation and the A/N bit disagree, then a memory mapped I/O operation was performed and an exception occurs. When this happens, the Code Morphing process takes the necessary actions to correct the access type of the re-translation and a rollback of any operations performed. Then, the correct re-translation is performed, scheduling the memory access operations in-order [7].

In contrast to the Crusoe, the DAISY microprocessor attempts to detect and schedule all memory mapped I/O operations in-order. If an operation is not detected correctly, then the DAISY detects the operation when it is executed and correctly re-translates the affected operations again [4].

In addition to the A/N bit, another type of protection bit, called a T-bit, is provided in every address translation in the TLB. The T-bit helps to guard against the affects of self-modifying code by specifying for which memory pages a translation exists. If self-modifying code is encountered, then the corresponding translation(s) must be invalidated and the new code re-translated prior to the code segment being executed again [7].

If a memory write occurs on a memory page for which the corresponding T-bit in the TLB is set, then an exception occurs and the microprocessor invalidates the translation(s). When the corresponding Intel X86 code segment is to be executed again, the translator will be called and the code segment will be re-translated. This mechanism can also be used to specify which memory pages the re-translations depend upon not being accessed by write operations [7]. The DAISY microprocessor supports self-modifying code by including a special “read-only” bit in every unit of memory [4].

3.5. Exception Handling

When an X86 exception is detected, the Crusoe microprocessor must ensure that the state of the emulated X86 microprocessor is exactly the same as it would be in the Intel X86 microprocessor. This is accomplished by rolling back the state of the working registers by copying the values located in the official registers into the working registers if needed and removing any uncommitted stores from the gated store buffer. Once the uncommitted stores are removed from the gated store buffer, the value in the register holding the tail pointer of the buffer is replaced with the value stored in the gate pointer of the buffer [7].

After the state of the working-values and the gated store buffer are successfully rolled-back, the state of the emulated microprocessor is the same as it would be in the Intel X86 microprocessor at the beginning of the code segment which caused the exception. Next, the Code Morphing process re-translates, re-executes, and commits the results of re-translation of each X86 instruction (in-order) until the exception occurs again. Once the exception occurs, the official state of the microprocessor is correct and the exception can be properly handled [7].

At the end of this process, the resulting re-translations can be stored into the translation buffer because they correctly handle the exception. Then, if the exception occurs in the future, the system will correctly handle the exception without having to re-translate the associated X86 code again [7].

3.6. Summary

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.

Transmeta has successfully used dynamic translation and execution of programs to reduce the power requirements of the Crusoe microprocessor by 60%-70% over compatible microprocessors [14]. This has been accomplished by reducing the complexity of the underlying hardware architecture of the microprocessor [7].

The Crusoe is innovative in its combination of simplistic hardware and the dynamic translation process to create a system capable of emulating another microprocessor. This innovative approach is a new direction in microprocessor design that may allow designers to create more sophisticated microprocessors that are compatible with legacy microprocessors.

4. Comparison of the DAISY and Crusoe Microprocessors

4.1. Overview

Both the DAISY and Crusoe microprocessors are static microprocessors that are completely compatible with the binary machine code of other microprocessors. These two microprocessors utilize the dynamic translation process of Figure 3 to re-translate program code compiled for different microprocessors into machine code for their own VLIW processor core. In both microprocessors, if a re-translated portion of machine code is frequently executed, it is optimized [6, 7].

Although the DAISY and Crusoe appear to be similar from a high-level viewpoint, their detailed implementations are quite different, as shown in Table 2.

- The DAISY has been developed for research purposes [6], while the Crusoe has been developed as a commercial product [7].
- They use different approaches to re-translation and optimization of machine code.
- The DAISY microprocessor uses optimizations that center around branch analysis [6].
- The Crusoe microprocessor performs specialized optimizations only valid on Intel X86 machine code [7].

Because the DAISY and Crusoe are designed to be compatible with different microprocessors, comparison of the two is nor as straightforward as it would have been otherwise. An overview of some similarities and differences between the two microprocessors has been presented in this subsection. In the next subsection, the difficulty encountered in making direct comparisons of certain aspects of these two microprocessors is presented. Subsection 4.3 presents a comparison between the architecture of the two microprocessors, followed by a comparison of how they re-translate machine code in Subsection 4.4. After this, the differences in the approach taken to scheduling by the two microprocessors are presented in Subsection 4.5, followed by a look at the different optimizations utilized by the microprocessors in Subsection 4.6. Finally, the different ways in which the microprocessors handle special situations (e.g., self-modifying code) is presented in Subsection 4.7.

Table 2. A comparison summary of the DAISY and Crusoe microprocessors.

	DAISY	Crusoe
Objective:	Research Based	Commercial Based
Goal:	Completely compatible with an existing microprocessor	Completely Compatible with the Intel X86 Less expensive
Architecture:	VLIW Based Clustered Official PowerPC Registers Clustered Cache Support for profiling Tree-based Instructions	VLIW Based Hardware and Software Based Official X86 Registers “Gated Store Buffer” Specialized TLB
Translation:	Goal: High ILP with low overhead cost due to compilation Performs an initial re-translation on the first occurrence of a PowerPC instruction Further re-translates and optimizes code with moderate and aggressive approaches if necessary Performs code profiling Uses tree groups as unit of translation	Performs re-translation on first occurrence of an X86 instruction Further retranslates code with less speculation and optimization if necessary
Scheduling:	Speculative load operations Memory Mapped I/O not speculatively scheduled	Speculative Memory Operations
Optimization:	Copy Propagation Load Store Telescoping Branch Analysis	Removes X86 Segmentation Process Removes memory boundary checks Sub-expression elimination Removal of commit operations Register Renaming Code Motion Data Aliasing Copy Elimination Use of alias hardware
Provides for:	Exception Handling Self-Modifying Code Self-Referential Code	Exception Handling Self-Modifying Code

4.2. Difficulties in Comparing the DAISY and Crusoe Microprocessors

Comparing the performance of the DAISY and Crusoe microprocessors is difficult because they are targeted at emulating different microprocessors. The DAISY system emulates the RISC-like PowerPC microprocessor [6] and the Crusoe microprocessor emulates the CISC-like Intel X86 microprocessor [7].

Adding to the difficulties in comparing these two microprocessors is the lack of information on the performance of the Crusoe microprocessor. This is due to Transmeta's objective of marketing their microprocessor based on power consumption rates as opposed to performance in terms of speed. Even if a computer system based on the Crusoe was purchased and benchmarked, there is no computer system based on the DAISY to compare it against because the DAISY has never been fabricated [6]. Additionally, if a DAISY microprocessor was available, other factors would impede a true comparison, e.g., peripheral hardware support.

In light of the difficulties discussed in this subsection, it is interesting to see how the DAISY works compared to the Crusoe. The designers of the DAISY have a research focus and are interested in furthering the state of microprocessor emulation and exploring novel techniques [6], while the Crusoe was designed from the beginning to be a commercial product [7]. By analyzing the design of the Crusoe, it is informative to note the techniques the designers chose as opposed to those chosen by the designers of the DAISY.

4.3. The Architectures of the DAISY and Crusoe Microprocessors

The VLIW processor core found in the DAISY is clustered and supports tree-based instructions. This type of processor core relies heavily on branch analysis and works well with programs in which there are several different paths for the flow of control to follow [6]. Thus, the microprocessor may be well suited for decision-based applications (e.g., artificial intelligence applications), but perhaps less suited for computationally intense applications (e.g., matrix multiplication). The DAISY microprocessor is designed not only for emulating the PowerPC microprocessor, but also as a base microprocessor that can be modified and extended to emulate any conventional microprocessor [6]. In contrast to this, the Crusoe microprocessor is designed only with the intent of emulating the Intel X86 microprocessor [7].

Transmeta has not released information that specifies the level of sophistication of the VLIW processor core of the Crusoe. Instead, they have provided information explaining the gated store buffer, TLB (which serves the same purpose as the instruction cache used by DAISY), and translation buffer found in the microprocessor. The gated store buffer represents a novel hardware approach to updating the state of the emulated microprocessor. This approach preserves the official state of the system and allows an automatic rollback of the unofficial state to the official state when needed [7], whereas the DAISY must manually rollback any writes to memory.

Additionally, the Crusoe provides shadow registers for the official X86 registers simplifying the rollback of the registers. In contrast, the DAISY uses extra registers to store changes to the state of the emulated microprocessor and copies the values to the official registers in the order that the instructions were to be executed in the original machine code making the rollback of the registers more complex.

Because Transmeta has not released much information detailing the architectural aspects of the Crusoe, it is difficult to determine the precise level of difference between its architecture and that of DAISY. Because the Crusoe is designed specifically to emulate the Intel X86 microprocessor, it might be expected that the VLIW core of the Crusoe resembles the processor core of the X86 microprocessor.

4.4. The Re-translation Processes

The basic approach taken to re-translation in the DAISY and Crusoe microprocessors is similar. They both convert machine code compiled for the emulated microprocessor into machine code for a VLIW-based processor core. When a segment of code is re-translated, they both immediately convert the code into the appropriate operations and store them for optimization at a later time. In both of the microprocessors, this process is transparent to the machine code being executed on the emulated microprocessor.

Although both of the microprocessors re-translate code the first time it is encountered, they approach the process differently. The DAISY not only re-translates machine code, but it also builds tree groups and collects profiling information on the machine code to help guide optimization at a later time [6]. In contrast, instead of using tree groups, the Crusoe re-translates machine code one instruction at a time and performs optimizations on straight-line segments of code [7] and there is no indication that it uses an instruction format similar to the tree-based instructions of DAISY.

As both microprocessors re-translate code, they link the translations together to remove calls to the re-translation process as much as possible. This helps to speed up the execution of re-translated code. Once re-translation has occurred on a segment of code, both of the microprocessors optimize the segment after it has been determined as a frequently executed segment of code [6, 7].

The DAISY has more steps in the re-translation process of code and it employs a more sophisticated approach to re-translating code by using tree groups as the unit of translation [6]. Even though the re-translation process of the DAISY system is more complex than that of the Crusoe microprocessor, the Crusoe has shown that such sophistication is not mandatory in implementing a commercially viable microprocessor that employs dynamic translation.

4.5. Scheduling Re-translated Operations

Both the DAISY and Crusoe microprocessors perform speculative scheduling of certain types of operations. The main difference in their approaches is that the Crusoe speculatively schedules all memory operations [7], while the DAISY only speculatively schedules memory stores that do not affect memory mapped I/O devices [6]. The DAISY detects most memory mapped I/O operations at re-translation time and schedules them in-order [6], whereas the Crusoe detects such operations at run-time and then re-translates the affected code, scheduling the memory mapped I/O operations in-order [7].

4.6. Optimization of Re-translated Machine Code

The optimizations used in the DAISY and Crusoe microprocessors vary greatly. The optimizations that DAISY performs (as specified in [6]) are copy propagation and load-store telescoping [6]. These optimizations are fairly generic in that they can be applied to sequences of DAISY operations no matter what microprocessor the DAISY system is targeted to emulate.

The optimizations performed by the Crusoe contain generic and system specific optimizations. Among the generic optimizations are sub-expression elimination, removal of commit operations, register renaming, code motion, data aliasing, copy elimination, and aliasing of hardware [7]. In DAISY, copy propagation is used to perform the same optimization as register renaming; and load-store telescoping is similar to the Crusoe’s use of alias hardware.

The generic optimizations performed by the Crusoe give it an edge over DAISY in regards to the efficiency of compiled code. However, the branch analysis approach of DAISY may negate the potential performance advantages that the Crusoe has over DAISY due to these optimizations.

The Crusoe also performs optimizations that are specific to the Intel X86 machine code. These optimizations are the removal of the X86 segmentation process and the removal of upper boundary memory checks [7]. These optimizations help the Crusoe to compete with the Intel X86 microprocessor, but if the Crusoe was to be re-targeted to emulate a different microprocessor, some of these may become invalid.

4.7. Handling of Special Situations

The DAISY and Crusoe microprocessors both provide facilities for handling special situations, e.g., self-modifying code. The microprocessors both handle PowerPC (DAISY) and Intel X86 (Crusoe) exceptions by ensuring the state of the emulated microprocessor is correct, performing any required actions, and then allowing the operating system to handle the exceptions as needed. If the exceptions are DAISY or Crusoe specific, then they are handled internally [6, 7].

The DAISY provides facilities for dealing with both self-modifying code and self-referential code [6], while (according to [7]) the Crusoe microprocessor only addresses self-modifying code. The mechanism used for handling such code is similar in both microprocessors.

The DAISY handles self-modifying code by adding a “read-only” bit to every memory unit allocated to the emulated PowerPC microprocessor. This “read-only” bit tells the DAISY VMM if the memory location should be invalidated due to a store to the memory location it occupies [6]. Similarly, the Crusoe handles self-modifying code by adding a T-bit to every address in the TLB. Then, if an address with its T-bit set is written to by the system, the Crusoe can handle the situation accordingly [7].

4.8. Summary

The designers of the Crusoe microprocessor started out with a basic VLIW processor core (that possibly resembles the core of the Intel X86 microprocessor’s multiple-issue execution unit) and added only the most necessary hardware components to this core to make the microprocessor run as fast and as power aware as possible [7]. On the other hand, the architects of the DAISY microprocessor chose to build their design on a sophisticated clustered tree-based VLIW processor core, to which they added a clustered cache system [6]. Although this design works well on programs that benefit from branch analysis, it is complex and may require more hardware to implement compared to the Crusoe microprocessor.

5. Proposed Future Research Directions

5.1. Overview

Several research projects have attempted to harness and capitalize on the flexibility of reconfigurable hardware, often realizing significant performance improvements for many target applications such as DNA matching, target recognition, pattern searching and encryption/decryption [15, 16, 17]. However, even as impressive as these performance improvements are, computing using reconfigurable hardware (often referred to as configurable or reconfigurable computing) has only become a custom solution to a relatively small set of problems and has yet to make a significant impact in the general purpose computing community [17]. This has led to an effort to merge general-purpose computing and reconfigurable hardware.

This section presents two research ideas that merge general-purpose microprocessors and reconfigurable hardware. The first idea presented is an architectural approach to use reconfigurable hardware in a microprocessor that performs dynamic translation (i.e., a DAISY-like microprocessor).

The second research idea is to analyze the instruction set of a current microprocessor. The goal of this analysis is to develop an analytical approach to designing microprocessors that employ reconfigurable hardware. In this work, instruction set analysis is performed using instruction set partitioning. Instruction set partitioning classifies instructions based on how frequently and how closely they are executed with respect to other instructions. If partitions exist, then it is conceivable to provide configurations for reconfigurable hardware that best match the characteristics of the instructions in each partition.

In the next subsection, an overview of the research idea of combining a DAISY-like microprocessor and reconfigurable hardware is presented. Subsection 5.3 presents an approach to instruction set partitioning, an overview of a microprocessor that could make use of instruction set partitions, and preliminary research that has been performed in this area.

5.2. An Architecture to Support Dynamic Translation with Reconfigurable Computing

In order for a microprocessor to be marketable and able to compete with current microprocessors, it must be compatible with a successful microprocessor in today's microprocessor market [7]. This is the motivation behind the Crusoe and DAISY microprocessors.

The DAISY and Crusoe microprocessors have made breakthroughs in emulating microprocessors at a hardware level using a dynamic translation process that is transparent to the applications running on the microprocessor. However, they must re-translate program code compiled for the emulated microprocessor. The goal of the optimizations and scheduling approaches used in the translation process is to execute the re-translated machine code fast enough to negate the effects of having to re-translate the initial machine code. It may be possible to further speed up the re-translation process by implementing the algorithms used in these processes and the resulting machine code (in certain circumstances) in reconfigurable hardware.

This concept follows closely to what has been done in the DISC (Dynamic Instruction Set Computer) microprocessor [18]. In DISC, each instruction is implemented as a stand-alone circuit module. Then, as the instructions are executed, the circuit for the current instruction is configured into the reconfigurable hardware and executed [18]. A similar approach can be taken to implementing the re-translation processes in reconfigurable hardware. The re-translation process can be synthesized into circuits and then partitioned into segments that can be implemented in reconfigurable hardware. When a segment is finished and control needs to pass to the next segment of the process, the reconfigurable hardware is re-configured to implement the new segment. To illustrate, the segmentation of a simple dynamic translation process is shown in Figure 23.

Another improvement, which may be beneficial to a microprocessor that uses dynamic translation to emulate an already existing microprocessor, is to add a reconfigurable execution unit to the core of the microprocessor, as shown in Figure 24. In this approach, the algorithm that controls the re-translation, optimization, and execution of the program code can determine to optimize an instruction or set of instructions by synthesizing them into circuits and implementing them in the reconfigurable execution unit. With an arrangement of execution units similar to that shown in Figure 24, the microprocessor can implement instructions in both static hardware and reconfigurable hardware.

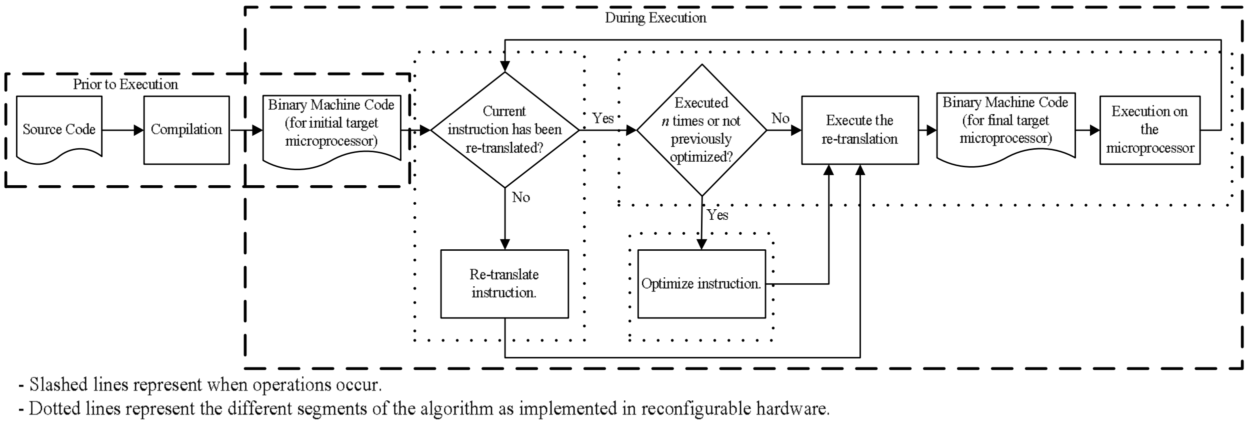

During Execution
Prior to Execution
Source Code
Compilation
Binary Machine Code (for initial target microprocessor)
Current instruction has been re-translated?
Yes
No
Executed n times or not previously optimized?
No
Yes
Execute the re-translation
Binary Machine Code (for final target microprocessor)
Execution on the microprocessor
Re-translate instruction.
Optimize instruction.
- Slashed lines represent when operations occur.
- Dotted lines represent the different segments of the algorithm as implemented in reconfigurable hardware.

Figure 23. The segmentation of a simple dynamic translation process in which each segment represents a different configuration of the same reconfigurable hardware.

Several mechanisms can be used to determine if an instruction or set of instructions should be synthesized and targeted for reconfigurable hardware. A simple mechanism for making this determination is an analysis of how often the instruction(s) are executed and how much time the microprocessor spends in the associated segments of program code. However, the time taken to synthesize the instructions into circuits and the time required to re-configure the reconfigurable execution unit must be considered in such a determination.

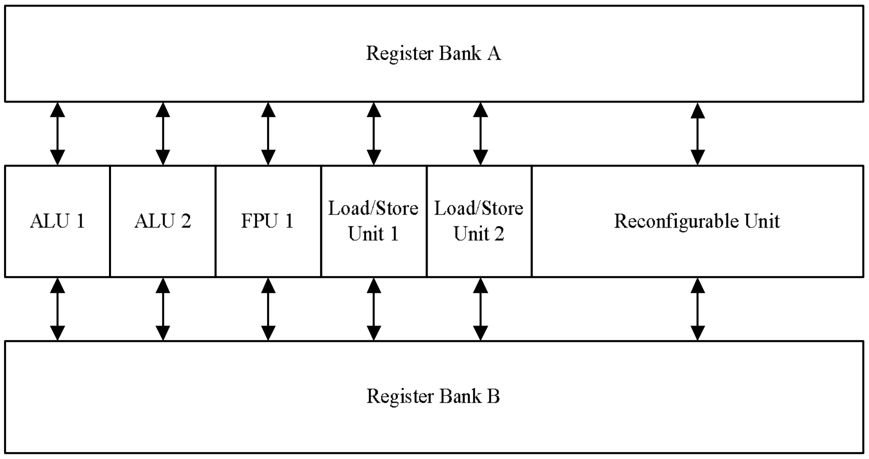


Figure 24. A microprocessor core that includes a reconfigurable execution unit.

A system that uses the concepts of implementing the re-translation process in reconfigurable hardware and/or synthesizing certain instructions or groups of instructions into reconfigurable hardware may perform better than microprocessors such as the DAISY or Crusoe that rely only on static hardware. The determination of this performance is likely to be associated with the time required to reconfigure the reconfigurable logic in the microprocessor and the time required to synthesize instructions into hardware.

With the modifications presented in this subsection, the system architecture of the resulting microprocessor would resemble Figure 25. These approaches may prove infeasible with current technology, due to the reconfiguration times of existing reconfigurable technology. However, reconfigurable technology continues to increase in logic capacity and decrease in reconfiguration time. Due to these ongoing improvements, the future may provide opportunities to implement such a system for general-purpose computing.

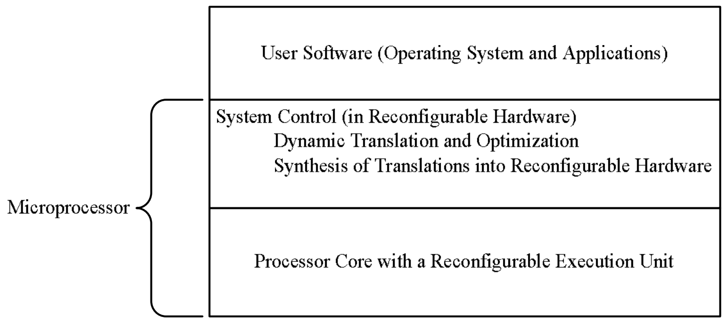

User Software (Operating System and Applications)
System Control (in Reconfigurable Hardware)
Dynamic Translation and Optimization
Synthesis of Translations into Reconfigurable Hardware
Microprocessor
Processor Core with a Reconfigurable Execution Unit

Figure 25. A high-level view of a system that uses dynamic translation and reconfigurable hardware.

5.3. Instruction Set Analysis

5.3.1. Overview

The research idea presented in this section is in the area of instruction set analysis and partitioning. The motivation behind instruction set analysis is to develop a formal approach to microprocessor design that is soundly based in a mathematical context. This presentation of instruction set partitioning is based on how closely instructions are executed with respect to each other.

The existence of instruction set partitions in existing instruction set architectures may be useful in the design of new microprocessors (based on reconfigurable hardware) or reconfigurable execution units for use within a microprocessor (e.g., Figure 24). A microprocessor (or execution unit) designed around instruction set partitions implements one partition of the instruction set in circuitry at a time. Instructions that are encountered that are outside of the partition that is currently supported can be emulated using the instructions found in the current partition. Or, when the control process of the microprocessor determines that the configurable hardware needs to be changed (i.e., because the partition that the majority of the instructions being executed by the current program are found in has changed), then the microprocessor re-configures to implement the appropriate partition in circuitry.

The drawback of using instruction set partitions (where only one partition is supported in hardware at a time) is that the execution of instructions that are not implemented in the current partition (that are executed using emulation) may be inefficient. However, the use of emulation allows any instruction (in the instruction set) to be performed without reconfiguring to implement the correct partition, bringing a sense of completeness to each partition if it can be used to emulate any instruction in the instruction set that it does not directly support in hardware.

Instruction set partitions may prove to be a very powerful tool in designing new instruction sets if instruction set partitions can be discovered and ways of analyzing them can be developed. Before the use and design of techniques to create instruction set partitions are explored in detail, an initial study has been performed to determine if instruction set partitions exist in the instruction sets of today's microprocessors. This section presents an initial study of an existing instruction set to see if instruction set partitions exist. The next subsection presents a discussion of a technique that may be able to detect such partitions in existing instruction sets along with preliminary experiments performed to verify this technique. Finally, the overall results of these experiments and areas for future work are presented in Subsection 5.3.3.

5.3.2. Detecting Instruction Set Partitions with Clustering

5.3.2.1. Overview

In this initial study, clustering was used to detect instruction set partitions for the Intel IA-32 instruction set [1]. Clustering is used because such techniques can determine if a set of data is weakly or strongly differentiated indicating if one or multiple classifications for the data exists [19]. This discovery of classifications of data (in our case instructions) is the purpose of this initial study.

For this study, the K-Means clustering technique was chosen because of its simplicity and ease of implementation. In the K-Means clustering algorithm (shown in Figure 26) the data is grouped into a set of mutually exclusive clusters, each of which has a center. Before the first iteration of the algorithm, the first *k* data units are chosen as the centers of *k* clusters (the *k* centers can also be randomly chosen). Then in step two, the remaining data is assigned to clusters dependent on the cluster that the data is closest to (this is determined by the distance between the data and the center of each cluster). In step three, each cluster is examined to determine the data point that should be the center of each cluster. Finally, the algorithm iterates steps two and three until convergence is reached [19]. In this study, convergence is determined by the distance between the old centers and the new centers. Convergence is reached when this distance becomes stable or cyclic.

1. Initialize centers to K random instructions that were executed.
2. Assign instructions to clusters.
3. Find a new set of centers.
4. Repeat steps 2 and 3 until the clusters converge.

Figure 26. The K-Means clustering algorithm derived from [19].

The Intel IA-32 instruction set [1] was chosen for this study because it is possible to trace the execution of a program on an Intel IA-32 microprocessor (e.g., Intel Pentium 4) [1] at the assembly language level one instruction at a time. This is possible using the *ptrace* system call [20] that is supported under the Linux operating system (e.g., Red Hat Linux 7.3). This allows a program to *exec* (i.e., start) another program to be traced and then attach to the program and control the execution of the program [20]. As each instruction is executed, it can be read from memory using the *ptrace* interface and disassembled into an assembly-level operation. The use of *ptrace* allows a finer granularity of control than other methods, such as using debuggers.

The GDB debugger [21] was also studied for this project. However, such debuggers only trace programs at the system call level and then disassemble large portions of the program at a time [21]. Therefore, it is impossible to know exactly what instructions in these portions were executed.

The rest of this section presents a series of three clustering experiments that were performed on the IA-32 instruction set as part of this study. In the next subsection, a description of the clustering experiments is presented, followed by a presentation of the results of the three experiments.

5.3.2.2. Clustering Experiments Performed

These experiments analyze the instructions executed by the POV-Ray (Persistence of Vision Raytracer) raytracing program [22]. This program was chosen because it uses sophisticated algorithms and can be successfully traced at the assembly-level using the technique discussed in Subsection 5.3.2.1. In these experiments POV-Ray was statically compiled for an Intel Pentium 4 system running the Red Hat Linux operating system (version 7.3), and was used to create the image shown in Figure 27. The input file, simple.pov, used for creating this image comes as part of the distribution packages of POV-Ray.

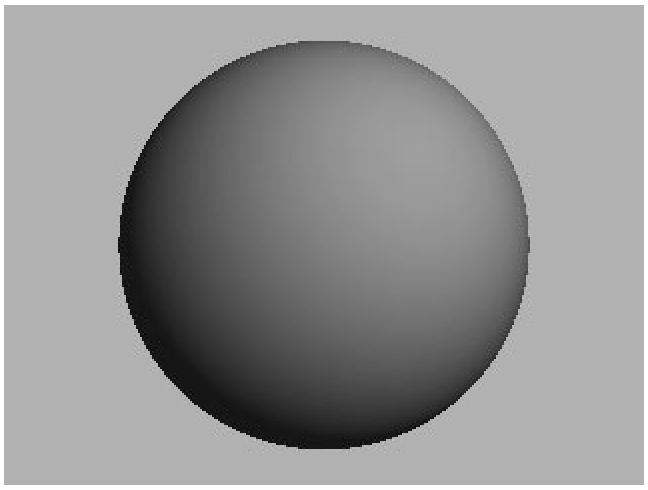


Figure 27. The image created using POV-Ray for the clustering experiments.

In order to perform the clustering, a distance metric is required to determine the cluster that instructions should be assigned to and the instructions that represent the resulting centers. This distance metric is crucial to the quality of the resulting clusters because it impacts how the instructions are classified into different clusters. The distance metric developed for these experiments measures of how close (in time) instructions are to other instructions in the instruction set for the program being executed.

The closeness of two instructions is based on how many delays (instructions) apart the two instructions are in the execution trace of the program. This information is collected as the program is being executed and is stored in a three-dimensional array. The array element associated with indices (*i*, *j*, *k*) represents the number of times instructions *i* and *j* are separated by a delay of *k*. A limit is placed on how far apart two instructions can be in the data collected. This limit serves two purposes: (1) the limit determines how close an instruction must be to the center of a cluster to be assigned to that cluster; and (2) the limit helps to keep the three-dimensional array stored in memory from growing large enough that this becomes an inefficient mechanism for collecting data. This limit is referred to as the *window size* because it effectively places a sliding *window* over the program for data to be collected (for these experiments, the size of the *window* is ten delays).

Once the delays for all the instructions executed by the program are calculated, the three-dimensional array is compressed into two dimensions by summing across the dimension that represents the different delays used (1-delay to *window size*-delays) and normalized by dividing each element of the resulting two-dimensional array by the sum of all of the elements. This results in a *likelihood* value that represents the probability of two instructions being encountered within the *window* used while executing the program. Finally, the distance between two instructions is calculated with the following distance metric: $distance(x,y) = 1 - likelihood(x,y)$.

Once these values were calculated for an execution of POV-Ray, the K-Means clustering algorithm (of Figure 26) was run three times to cluster the instructions, with ten clusters ($k = 10$). The results of the first run are discussed in the next subsection, followed by the results of the second and third runs. In these experiments, the clustering algorithm was allowed to iterate until convergence was observed (this occurred in less than 100 iterations for each experiment).

5.3.2.3. First Clustering

The results of the first run of the clustering algorithm are shown in Figure 28. In these results, a *distance to center* value of 1 indicates that the pair of the instruction and the instruction that is the center of the cluster was never encountered within the *window* used to collect delay data; and a *distance to center* value of -1 indicates that the instruction is the center of the cluster. Thus, a *distance to center* value close to 1 indicates that the pair of the instruction and the center of the cluster are far apart and a *distance to center* value close to 0 indicates that they are close together.

In the results reported in Figure 28, the clusters found are not distinct because the instructions that form each cluster are far away from the center of the cluster. Additionally, the centers of the clusters that are found have a high execution frequency and are spread throughout the execution of the program. This indicates that the centers that were found are not true centers of a temporal portion of executed instructions. This leads to the next run of the clustering algorithm, where the top ten most frequently executed instructions were removed from consideration in the clustering process. This was done in an effort to let the clustering algorithm discover different centers that are more reasonable than those found in the first run of the algorithm.

5.3.2.4. Second Clustering

The results of the second run of the clustering algorithm are shown in Figure 29; and the instructions removed from consideration in the clustering process are listed in Figure 30. As in the first run of the algorithm, this run of the algorithm resulted in clusters that are not distinct (the instructions assigned to each cluster are far away from the center of the cluster). However, in these results the execution frequency of the centers is lower and closer to the instructions found in the clusters than in the first run. Due to these results, one more run of the algorithm was performed where the top fifty most frequently executed instructions were removed from consideration in the clustering process.

Cluster	Center	Instruction	Distance to Center	Frequency
0				
1				
2	mov	adc	0.999997878	455
		add	0.974808389	6966594
		and	0.987613055	2810445
		bsf	0.999999951	9
		bsr	0.999995856	695
		call	0.986570572	3584263
		cdq	0.999998453	230
		cld	0.997068666	661674
		cmovbe	0.999999785	51
		cmovc	0.999950061	8906
		cmovg	0.999998122	356
		cmovl	0.999987677	2017
		cmovle	0.999987115	1794
		cmovnc	0.999999963	7
		cmovns	0.999994436	1248
		cmovnz	0.999971215	4915
		cmovs	0.999999551	93
		cmovz	0.999785881	37187
		cmp	0.962424883	10307302
		cwde	0.999999969	10
		dec	0.991280393	1634748
		div	0.999930509	8026
		fadd	0.99760333	1258335
		fiadd	0.999999997	1
		fidivr	0.999999994	2
		fild	0.999674382	167052
		fimul	0.999253358	153603
		fist	0.999999903	30
		fistp	0.995452523	694021
		fisub	0.999999933	30
		fisubr	0.999827698	76801
		fld	0.975498895	10040418
		fldcw	0.987746379	1848902
		fldz	0.998925467	754305
		frndint	0.998793904	230400
		fsin	0.999999252	1000
		fstcw	0.994017138	924451
		fstp	0.986350166	5693399
		fsub	0.999555381	224364
		fsubp	0.999712098	153857
		fsubr	0.99838611	586078
		fucomp	0.999059471	371776
		fucompp	0.997744508	1418651
		fxch	0.993831682	5666274
		idiv	0.999999997	1
		imul	0.999983845	2371
		inc	0.978715052	6815934
		int	0.9999868	3306
		ja	0.995510163	696666
		jbe	0.994732295	1018722
		jc	0.997276148	944291
		jcxz	0.999993724	3931
		jg	0.998095061	489419
		jge	0.999353454	127657
		jl	0.998542979	292728
		jle	0.992674314	1852853
		jmp	0.993430533	2117456
		jnc	0.998507619	558893
		jns	0.995821084	905932
		jnz	0.974842644	7425393
		js	0.999424586	162646
		jz	0.976439243	6082623
		lea	0.978853692	6183661
		leave	0.998697305	329086
		mov	-1	55756233
		movsb	0.999996783	841
		movsd	0.999997451	560
		movsx	0.999801194	53906
		movzx	0.973076156	6549733
		mul	0.999984907	3455
		neg	0.999707893	66737
		nop	0.996016973	602617
		not	0.999998993	248
		or	0.998469698	274909
		pop	0.967416922	10073504
		push	0.931308873	20963662
		rdtsc	0.999999888	19
		ret	0.98768832	3584393
		sar	0.99981345	104440
		sbb	0.999998562	239
		scasb	0.999995694	2495
		seta	0.999999949	20
		setbe	0.999999925	20
		setc	0.999999968	13
		setg	0.999999934	10
		setnz	0.999946941	15097
		setz	0.999958229	7530
		shl	0.995603483	711986
		shld	0.999993842	1322
		shr	0.999781983	61838
		shrd	0.999999904	22
		sub	0.977096893	6602932
		test	0.965569166	8884945
		xchg	0.999996983	817
		xor	0.992608943	1520276
3				
4				
5				
6				
7	fmul	fabs	0.999779415	260260
		faddp	0.996015073	1522662
		fdiv	0.999851326	67423
		fdivr	0.999999955	10
		fdivrp	0.999708228	78853
		fidiv	0.999827701	153603
		fld1	0.999420888	722827
		fmul	-1	4780494
		fmulp	0.999624724	174073
		fsqrt	0.999574739	160442
		fst	0.998424392	906578
		fstsw	0.999100411	2511001
		fsubrp	0.99991912	158659
		fucom	0.999285063	719574
8	cmova	cmova	-1	60269
9				

Figure 28. Results of the first run of the K-Means clustering algorithm.

Cluster	Center	Instruction	Distance to Center	Frequency
0	rdtsc	bsf	1.000000000	9
		cmovc	1.000000000	8906
		cmovnc	1.000000000	7
		cmovns	1.000000000	1248
		cmovs	1.000000000	93
		fiadd	1.000000000	1
		fidivr	1.000000000	2
		fsin	1.000000000	1000
		idiv	1.000000000	1
		movsb	1.000000000	841
		rdtsc	-1.000000000	19
		sbb	0.999999987	239
		scasb	1.000000000	2495
		setbe	1.000000000	20
		setg	1.000000000	10
		shrd	1.000000000	22
1				
2	movzx	adc	0.999999997	455
		bsr	0.999999996	695
		call	0.999663031	3584263
		cdq	0.999999997	230
		cmova	0.999999904	60269
		cmovbe	0.999999952	51
		cmovg	0.999999481	356
		cmovl	0.999998016	2017
		cmovle	0.999999762	1794
		cmovnz	0.999997732	4915
		cmovz	0.999974410	37187
		cwde	0.999999993	10
		dec	0.998154598	1634748
		div	0.999999998	8026
		imul	0.999996971	2371
		int	0.999999995	3306
		ja	0.999380050	696666
		jbe	0.998355982	1018722
		jc	0.998317459	944291
		jcxz	0.999987202	3931
		jg	0.999648894	489419
		jle	0.997830772	1852853
		jmp	0.998801977	2117456
		jnc	0.998972709	558893
		jns	0.999552024	905932
		js	0.999993101	162646
		lea	0.999312506	6183661
		movsx	0.999982744	53906
		movzx	-1.000000000	6549733
		mul	0.999996291	3455
		neg	0.999973838	66737
		nop	0.999110540	602617
		not	0.999999932	248
		sar	0.999865645	104440
		seta	0.999999996	20
		setc	0.999999996	13
		setnz	0.999992463	15097
		setz	0.999993145	7530
		shl	0.999581110	711986
		shld	0.999999995	1322
		shr	0.999930318	61838
		xor	0.999383735	1520276
3				
4				
5	fxch	and	0.998178352	2810445
		cld	0.999806572	661674
		fabs	0.999658740	260260
		fadd	0.998431985	1258335
		faddp	0.995373855	1522662
		fdiv	0.999827419	67423
		fdivr	0.999999985	10
		fdivrp	0.999878210	78853
		fidiv	0.999597968	153603
		fild	0.999997048	167052
		fist	0.999999933	30
		fisub	0.999999933	30
		fisubr	0.999827700	76801
		fld1	0.998713842	722827
		fldz	0.999098082	754305
		fmul	0.990498765	4780494
		fmulp	0.999746315	174073
		fsqrt	0.999684537	160442
		fst	0.997792166	906578
		fstp	0.995227629	5693399
		fstsw	0.996091204	2511001
		fsub	0.999659803	224364
		fsubp	0.999712786	153857
		fsubr	0.998871646	586078
		fsubrp	0.999600079	158659
		fucom	0.998189202	719574
		fucomp	0.999754748	371776
		fucompp	0.998088327	1418651
		fxch	-1.000000000	5666274
		jge	0.999950691	127657
		jz	0.998933851	6082623
		leave	0.999963644	329086
		ret	0.999604647	3584393
6				
7	movsd	movsd	-1.000000000	560
		xchg	0.999999618	817
8				
9	fldcw	fimul	0.999655398	153603
		fistp	0.998157919	694021
		fldcw	-1.000000000	1848902
		frndint	0.999310802	230400
		fstcw	0.997583543	924451
		jl	0.999942746	292728
		or	0.999540535	274909

Figure 29. Results of the second run of the K-Means clustering algorithm.

mov	test
push	jnz
cmp	add
pop	inc
fld	sub

Figure 30. The ten most frequently executed instructions.

5.3.2.5. Third Clustering

In the final run of the clustering algorithm, the top fifty most frequently executed instructions (listed in Figure 31) were removed from consideration in the clustering process. The major result of this run of the clustering algorithm (as shown in Figure 32) is the same as in the first and second runs. This result is that the clusters that were found are not distinctive. The instructions assigned to each cluster are still far away from the centers of the clusters. Additionally, in this run, the centers of each cluster have execution frequencies that are close to that of the other instructions in the clusters. This indicates that the instructions that are spread through out the program (and are executed the most frequently) are those contained in the fifty instructions removed from consideration in the clustering process.

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 31. The fifty most frequently executed instructions.

5.3.3. Overall Results of the Experiments and Future Work

The three experiments conducted are inconclusive because the clusters found in all three runs of the clustering process (as discussed in Subsections 5.3.2.3, 5.3.2.4, and 5.3.2.5) are not distinctive from each other. There are several possible reasons for why this occurred: (1) using a clustering technique to discover instruction set partitions may be inappropriate; (2) the K-Means clustering technique may not be appropriate for this application of cluster analysis; (3) the distance metric used may be poorly formulated or may need to be revised; and/or (4) the clustering program used in these experiments may contain error(s). Before more work is done in this area, these issues will be addressed. Thus, more research into clustering and instruction set partitioning needs to be performed; and other clustering techniques besides the K-Means clustering algorithm (Figure 26) need to be investigated. Additionally, the distance metric used needs to be reviewed and tested; and the correctness of the clustering program needs to be verified. One possible way to test the distance metric and verify the correctness of clustering program, is to test them with an assembly language program that can be analyzed by hand to see if the results match.

Cluster	Center	Instruction	Distance to Center	Frequency
0	fidiv	adc	1.000000000	455
		cdq	1.000000000	230
		cmova	1.000000000	60269
		cmovbe	1.000000000	51
		cmovc	1.000000000	8906
		cmovg	1.000000000	356
		cmovl	1.000000000	2017
		cmovle	1.000000000	1794
		cmovnc	1.000000000	7
		cmovns	1.000000000	1248
		cmovs	1.000000000	93
		cwde	1.000000000	10
		fdivr	1.000000000	10
		fiadd	1.000000000	1
		fidiv	-1.000000000	153603
		fidivr	1.000000000	2
		fist	1.000000000	30
		fisubr	0.999885134	76801
		fsub	0.999885134	224364
		fsubp	0.999885134	153857
		idiv	1.000000000	1
		movsx	1.000000000	53906
		rdtsc	1.000000000	19
		sbb	1.000000000	239
		scasb	1.000000000	2495
		setbe	1.000000000	20
		setg	1.000000000	10
		shld	1.000000000	1322
1	fsqrt	fdiv	0.999975346	67423
		fdivrp	0.999941033	78853
		fmulp	0.999917909	174073
		fsqrt	-1.000000000	160442
		fsubrp	0.999963647	158659
2	bsr	bsr	-1.000000000	695
3	bsf	bsf	-1.000000000	9
		div	1.000000000	8026
4				
5	seta	seta	-1.000000000	20
		setc	0.999999989	13
6	shr	cmovnz	0.999998157	4915
		cmovz	0.999999982	37187
		imul	0.999999788	2371
		int	0.999999997	3306
		jcxz	0.999999412	3931
		jge	0.999999716	127657
		js	0.999999048	162646
		movsb	0.999998744	841
		movsd	0.999999164	560
		mul	0.999997220	3455
		neg	0.999999023	66737
		not	0.999999953	248
		sar	0.999999657	104440
		setnz	0.999999999	15097
		setz	0.999998119	7530
		shr	-1.000000000	61838
		shrd	0.999999991	22
		xchg	0.999999541	817
7	fimul	fimul	-1.000000000	153603
8	fild	fild	-1.000000000	167052
		fsin	0.999999253	1000
9	fisub	fisub	-1.000000000	30

Figure 32. Results of the third run of the K-Means clustering algorithm.

In summary, the results presented in this study are inconclusive and the experiments that were performed need to be scrutinized. Additionally, more work needs to be performed in the area of instruction set analysis; and different ways of evaluating instruction sets need to be formulated.

5.4. Summary

Two ways to improve the design of microprocessors have been presented in this section. The first deals with combining the dynamic translation process of Figure 3 with reconfigurable hardware to create a reconfigurable microprocessor that uses dynamic translation to execute programs. More research needs to be performed on the concept of combining dynamic translation and reconfigurable hardware. This concept is only in the initial stages of development and has not been pursued farther than proposing the ideas presented in Subsection 5.2. The second way to improve microprocessors is to analyze the current instruction sets being used in today's microprocessors and attempt to develop a methodology of how to evaluate the design and use of instruction sets.

The initial step in an analysis of current instruction sets using instruction set partitioning has been presented in this section. This study needs to be expanded further and the methods need to be reviewed at a deeper level.

6. Conclusions

This report 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. At the end of the report, two possible research directions were introduced: (1) reconfigurable computing combined with dynamic translation, and (2) instruction set partitioning analysis.

The two microprocessors reviewed in this report 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 greatly 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 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. These microprocessors harness 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.

At the end of the report, two concepts for improving on today’s microprocessors were presented. The first one is to combine the dynamic translation process with a reconfigurable microprocessor. Such a microprocessor may be able to outperform a static counterpart because it implements the dynamic translation process in hardware and the optimization process has the option of synthesizing an instruction or segment of code into circuits that can be implemented in the reconfigurable hardware. This implementation of instruction(s) in hardware could speed up execution of the associated operations.

The second idea presented at the end of this report is instruction set partitioning. This presentation discusses an initial study of what properties a good instruction set possesses. The ideas presented in this part of the report are pursued in hopes of being able to formalize the design and use of instruction sets and how they are evaluated. However, the results of this study are inconclusive.

More research needs to be performed in the areas of dynamic translation, reconfigurable computing, and the design of microprocessors and the instruction sets they implement. These areas represent vast opportunities in improving the microprocessors that are currently being developed and produced. As the feature size of the technologies used to implement microprocessors shrinks, they will become faster enabling more processing of machine code to be performed with less execution latency. Additionally, reconfigurable technology will be able to implement larger circuits and reconfigure the circuits implemented in less time as well. This will continue to open more opportunities for deeper research as these technological improvements are realized.

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/852569B20050FF778525699600719DF2, 2002.

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

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

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

[6] 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.

[7] 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*, February 2000.

[8] 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.

[9] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers Inc., San Francisco, California, 1996.

[10] 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.

[11] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1988.

[12] K. Ebcioğlu, E.R. Altman, S. Sathaye, and M. Gschwind, "Optimizations and Oracle Parallelism with Dynamic Translation," *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999, pp. 284-295.

[13] D.A. Patterson and J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, Morgan Kauffman Publishers, Inc., San Francisco, California, 1998.

[14] A. Klaiber, “The Technology Behind Crusoe Processors: Low-Power X86-Compatible Processors Implemented with Code Morphing Software,” Transmeta Corporation, Santa Clara, California, http://www.transmeta.com/about/press/white_papers.html, January 2000.

[15] J.R. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor,” Proceedings of the 5th Annual IEEE Symposium on Field Programmable Custom Computing Machine, 1997, pp. 12-21.

[16] J.M. Arnold, D.A. Buell, and E.G. Davis, “Splash 2,” Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1992, pp. 316-322.

[17] J. Villasenor and B. Hutchings, “The Flexibility of Configurable Computing,” IEEE Signal Processing Magazine, September 1998, Vo. 15, No. 5 , pp. 67-84.

[18] M.J. Wirthlin and B.L. Hutchings, “A Dynamic Instruction Set Computer”, *Proceedings of the 1995 IEEE Symposium on FPGAs for Custom Computing Machines*, 1995, pp. 99-107.

[19] M.R. Anderberg, *Cluster Analysis for Applications,* Academic Press, New York, New York, 1973.

[20] *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.

[21] *The GNU Project Debugger: Documentation for GDB version 5.2.1, GDB Internals*, Free Software Foundation, Inc, http://sources.redhat.com/gdb/download/onlinedocs/gdbint.html, April 2002.

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