Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 4, April 2014, pg.944 – 953

RESEARCH ARTICLE

V-ISA use in Transmeta Crusoe Processor

Dipali M. Dhaskat[1], P. P. Karde[2]

[1] Computer Science and information technology, HVPM COET, Amravati, India

[2] Computer Science and information technology, HVPM COET, Amravati, India

[1] dipali.dhaskat@gmail.com; [2] p_karde@rediffmail.com

Abstract— A virtual instruction set architecture (V-ISA) implemented via a processor-specific software translation layer can provide great flexibility to processor designers .Recent examples such as DAISY and Crusoe. Crusoe is the new microprocessor which has been designed especially for the mobile computing market. This microprocessor was developed by a small Silicon Valley start-up company called Transmeta Corp in the January 2000. After five years of secret toil at an expenditure of $100. This processor was based on the x86 architecture with a software layer called Code Morphing Software(CMS) comprised of an interpreter, a run time system, and code optimizer running on top of the processor. Crusoe is the first processor whose instruction set is implemented in the software; the benefit of that being - the software could "learn" the behaviour of a program as it runs, improving with time by recognizing patterns previously encountered and making smart decisions based on those patterns, thus making it the first "smart" processor. The key to the Crusoe processor is in the CMS which explores a unique approach called commit and roll back supported by the hardware along with translation and efficient interpretation of instructions. As a result of CMS, Crusoe offers out-of-the box compatibility to most operating system as well as an attractive choice for mobile computing.

Keywords— CMS; Rollback; Atom; Shadow; Molecule

I. INTRODUCTION

Mobile computing has been the buzzword for quite a long time. Mobile computing devices like laptops, web slates & notebook PCs are becoming common nowadays. The heart of every PC whether a desktop or mobile PC is the microprocessor .Several microprocessors are available in the market for desktop PCs from companies like Intel, AMD and Cyrix The heart of every PC whether a desktop or mobile PC is the microprocessor .Several microprocessors are available in the market for desktop PCs from companies like Intel, AMD and Cyrix etc.
The mobile computing market has never had a microprocessor specifically designed for it. Crusoe is the new microprocessor which has been designed especially for the mobile computing market.

The concept of Crusoe is well understood from the simple sketch of the processor architecture, called 'amoeba'. In this concept, the x86-architecture is an ill-defined amoeba containing features like segmentation, ASCII arithmetic, variable-length instructions etc. The amoeba explained how a traditional microprocessor was, in their design, to be divided up into hardware and software. Thus Crusoe was conceptualized as a hybrid microprocessor that is it has a software part and a hardware part with the software layer surrounding the hardware unit. The role of software is to act as an emulator to translate x86 binaries into native code at run time. Crusoe is a 128-bit microprocessor fabricated using the CMOS process. The chip's design is based on a technique called VLIW to ensure design simplicity and high performance. Besides this it also uses Transmeta's

two patented technologies, namely, Code Morphing Software and Long run Power Management. It is a highly integrated processor available in different versions for different market segments.

II. VIRTUAL INSTRUCTION SET COMPUTERS

As a step towards loosening these restrictions, several research and commercial groups have advocated a class of architectures we term Virtual Instruction Set Computers (VISC) . Such an architecture defines a virtual instruction set that is used by *all user and operating system software*, An implementation of the architecture includes both (a) a hardware processor with its own instruction set (the implementation ISA or IISA), and (b) an *implementation-specific* software translation layer that translates virtual object code to the I-ISA. Because the translation layer and the hardware processor are designed together, Smith et al. refer to this implementation strategy as a *code signed virtual machine* [32]. Fisher has described a closely related vision for building families of processors customized for specific application areas that maintain compatibility and performance via software translation [15].

At the most basic level, a VISC architecture decouples the program representation (V-ISA) from the actual hardware interface (V-ISA), allowing the former to focus on capturing program behaviour while the latter focuses on software control of hardware mechanisms. This brings two fundamental benefits to the hardware processor design and its software translation layer:

1) The virtual instruction set can include rich program information not suitable for a direct hardware implementation, and can be independent of most implementation-specific design choices.
2) The I-ISA and its translator provide a *truly cooperative* hardware/software design: the translator can provide information to hardware through implementation specific mechanisms and instruction encodings, while the hardware can expose novel micro architectural mechanisms to allow cooperative hardware/software control and also to assist the translator.

These two fundamental benefits could be exploited in potentially unlimited ways by processor designers. Prior work has discussed many potential hardware design options enabled by the VISC approach, which are impractical with conventional architectures. Furthermore, I-ISA instruction encodings and software-controlled mechanisms can both be changed relatively easily with each processor design, something that is quite difficult to do for current processors. Cooperative hardware-software techniques (including many examples proposed in the literature) can become much easier to adopt. Finally, external compilers can focus on machine-independent optimizations while the translator serves as a common back-end customized for the processor implementation. The cost of this increased flexibility is the possible overhead of software translation (*if* it must be done "online").Nevertheless, recent advances in dynamic compilation, program optimization, and hardware speed can mitigate the performance penalty, and could make this idea more viable today than it has been in the past. Furthermore, hardware mechanisms can be used to assist these tasks in many ways

A. Our Contribution: Design for A Virtual Instruction Set

Although virtual architectures have been discussed for a long time and real implementations exist (viz., IBM S/38 and AS/400, DAISY, and Transmeta's Crusoe), there has been little research exploring *design options* for the V-ISA. Both DAISY and Crusoe used traditional hardware ISAs as their V-ISA. The IBM machines do use a specially developed V-ISA, but, as we explain in Section 6, it is also extremely complex, OS-dependent, requires complex OS services for translation. The OS-dependent design enables a translation strategy integrated with the OS, which is impractical with

OS-independent designs. We believe a careful design for the V-ISA *driven by the needs of compiler technology*, yet "universal" enough to support *arbitrary* user and OS software, is crucial to achieve the full benefits of the virtual architecture strategy. A common question is whether Java byte code (as suggested by Smith et al. [32]) or Microsoft's Common Language Infrastructure (CLI) could be used as a V-ISA for processors. Since a processor V-ISA must support *all external user software and arbitrary operating systems*, we believe the answer is "no". These representations are designed for a certain class of languages, and are not sufficiently language independent for a processor interface. They include complex runtime software requirements, e.g., garbage collection and extensive runtime libraries, which are difficult to implement without operating system support. Finally, they are generally not well-suited for low-level code such as operating system trap handlers, debuggers, and performance monitoring tools. This paper proposes a virtual ISA called Low-level Virtual Architecture (LLVA), and an accompanying translation strategy that does not assume particular hardware support. More specifically, this work makes three contributions:

1) It proposes a V-ISA design that is rich enough to support sophisticated compiler analyses and transformations, yet low-level enough to be closely matched to native hardware instruction sets and to support all external code, including OS and kernel code.
2) It carefully defines the behaviour of exceptions and self modifying code to minimize the difficulties faced by previous translators for DAISY and Crusoe.

3) It describes a translation strategy that allows offline translation and offline caching of native code and profile information (unlike DAISY and Crusoe), by using an OS-independent interface to access external system resources.

In other work, we are developing a complete compiler framework for link-time, install-time, runtime, and offline ("idle-time") optimization on ordinary processors, based on essentially the same code representation . We discuss how the translation strategy proposed here can directly leverage those optimization techniques. The virtual instruction set we propose uses simple RISC like operations, but is fully typed using a simple language independent type system, and includes explicit control flow graphs and dataflow information in the form of a Static Single Assignment (SSA) representation. Equally important is what the V-ISA does *not* include: a fixed register set, stack frame layout, low-level addressing modes, limits on immediate constants, delay slots, speculation, predication, or explicit interlocks. All of these are better suited to the IISA than the V-ISA. Nevertheless, the V-ISA is low-level enough to permit extensive machine-independent optimization in source-level and link-time compilers (unlike Java byte code, for example), reducing the amount of optimization required during translation from V-ISA to I-ISA. The benefits of a V-ISA design can only be determined after developing new processor design options and the software/hardware techniques that exploit its potential. Our goal in this work is to evaluate the design in terms of its suitability as a V-ISA. We have implemented the key components of the compilation strategy for SPARC V9 and Intel IA-32 hardware processors, including an aggressive link time inter procedural optimization framework (which operates on the V-ISA directly), native code generators that can be run in either offline or JIT mode, and a software trace cache (for the SPARC V9) to support trace-based runtime optimizations. With these components, we address two questions:

1) Qualitatively, is the instruction set rich enough to enable both machine-independent and dependent optimizations during code generation?
2) Experimentally, is the instruction set low-level enough to map closely to a native hardware instruction set, and to enable fast translation to native code?

B. Design Goals for A Virtual ISA

Figure 1 shows an overview of a system based on a VISC processor. A VISC architecture defines an external instruction set (V-ISA) and a binary interface specification (V-ABI). An implementation of the architecture includes a hardware processor plus a translator, collectively referred to as the "*processor*." We use "*external software*" to mean all software except the translator. The translator is essentially a compiler which translates "virtual object code" in the V-ISA to native object code in the I-ISA. In our proposed system architecture, *all external software* may only use the V-ISA. This rigid constraint on external software is important for two reasons:

1) To ensure that the hardware processor can evolve both the I-ISA and its implementation details visible to the translator can be changed), without requiring any external software to be recompiled.
2) To ensure that arbitrary operating systems that conform to the V-ABI can run on the processor.

Because the primary consumer of a V-ISA is the software translation layer, the design of a V-ISA must be driven by an understanding of compiler technology. Most non-trivial optimization and code generation tasks rely on information about global control-flow, dataflow, and data dependence properties of a program. Such properties can be extremely difficult to extract from native machine code. The challenge is to design a V-ISA that provides such high-level information about program behaviour, yet is appropriate as an architecture interface for *all external software*, including applications, libraries, and operating systems. We propose a set of design goals for such a V-ISA:

1) Simple, low-level operations that can be implemented without a runtime system. To serve as a processor level instruction set for arbitrary software and enable implementation without operating system support, the V-ISA must use simple, low-level operations that can each be mapped directly to a small number of hardware operations.
2) No execution-oriented features that obscure program Behaviour: The V-ISA should exclude ISA features that make program analysis difficult and which can instead be managed by the translator, such as limited numbers and types of registers, a specific stack frame layout, low-level calling conventions, limited immediate fields, or low-level addressing modes.
3) Portability across some family of processor designs: It is impractical to design a "universal" V-ISA for all conceivable hardware processor designs. Instead, a good V-ISA design must enable some broad class of processor implementations and maintain compatibility at the level of virtual object code for all processors in that class (key challenges include bendiness and pointer size).
4) High-level information to support sophisticated program analysis and transformations: Such high-level information is important not only for optimizations but also for good machine code generation, e.g., effective instruction scheduling and register allocation. Furthermore, improved program analysis can enable more powerful cooperative software/hardware mechanisms as described above.

5) *Language independence*: Despite including high-level information (especially type information), it is essential that the V-ISA should be completely language.

III. **CRUSOE PROCESSOR VLIW HARDWARE**

A. Basic principles of VLIW Architecture

VLIW stands for Very Long Instruction Word. VLIW is a method that combines multiple standard instructions into one long instruction word. This word contains instructions that can be executed at the same time on separate different parts of the same chip or chips .This provides explicit parallelism. VLIW architectures are a suitable alternative for exploiting instruction-level parallelism (ILP) in programs that is, for executing more than one basic (primitive) instruction at a time. By using VLIW you enable the compiler, not the chip to determine which instructions can be run concurrently. This is an advantage because the compiler knows more information about the program than the chip does by the time code gets to the chip. These capabilities are exploited by compilers which generate code that has grouped together independent primitive instructions executable in parallel. The processors have relatively simple control logic because they do not perform any dynamic scheduling or reordering of operations as is the case in most contemporary superscalar processors. Dynamic scheduling is another important method when compiling VLIW code. The process, called split-issue splits the code into two phases, phase one and phase two. This allows for multiple instructions to execute at the same time. Thus, instructions that have certain delays associated with them can be run concurrently, and out-of-order execution is possible. The results computed in phase two are stored in temporary variables and are loaded into the appropriate phase one register when they are needed. VLIW has been described as *a natural successor to RISC*, because it moves complexity from the hardware to the compiler, allowing simpler, faster processors. The objective of VLIW is to eliminate the complicated instruction scheduling and parallel dispatch that occurs in most modern microprocessors. In theory, a VLIW processor should be faster and less expensive than a comparable RISC chip. The instruction set for a VLIW architecture tends to consist of simple instructions RISC like. The compiler must assemble many primitive operations into a single "instruction word" such that the multiple functional units are kept busy, which requires enough instruction-level parallelism (ILP) in a code sequence to fill the available operation slots.

B. VLIW in Crusoe Microprocessor

With the Code Morphing software handling x86 compatibility, Transmeta hardware designers created a very simple, high-performance, VLIW engine with two integer units, a floating point unit, a memory (load/store) unit. A Crusoe processor long instruction word, called a molecule, can be 64 bits or 128 bits long and contain up to four RISC-like instructions, called atoms. All atoms within a molecule are executed in parallel, and the molecule format directly determines how atoms get routed to functional units; this greatly simplifies the decode and dispatch hardware.

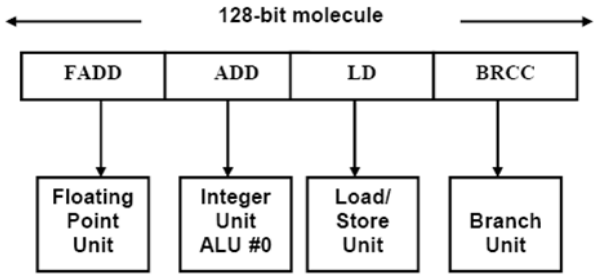

Fig. A molecule can contain up to four atoms, which are executed in parallel

. Fig shows a sample 128-bit molecule and the straightforward mapping from atom slots to functional units. Molecules are executed in order, so there is no complex out-of-order hardware. To keep the processor running at full speed, molecules are packed possible with atom. Code Morphing software allocates some of these to hold x86 state while others contain state internal to the system, or can be used as temporary registers, e.g., for register renaming in software. In the assembly code examples in this paper, we write one molecule per line, with atoms separated by semicolons.

IV. CODE MORPHING

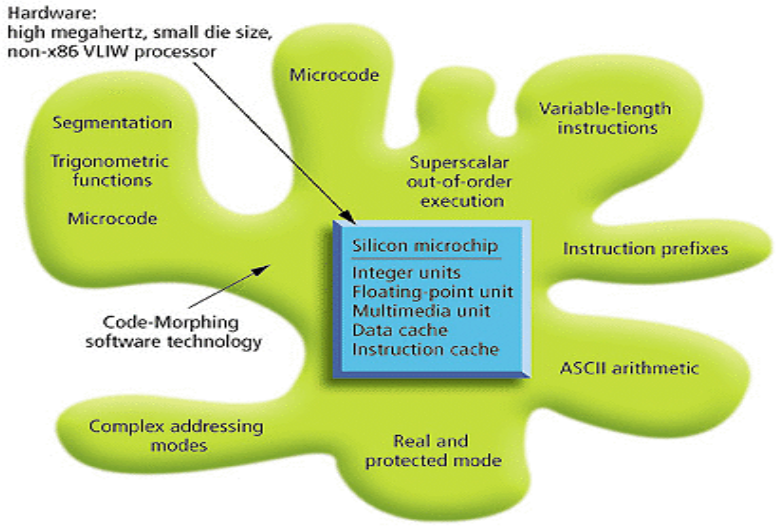

Fig : Crusoe- "AMOEBA"

The Transmeta designers have decoupled the x86 instruction set architecture (ISA) from the underlying processor hardware, which allows this hardware to be very different from a conventional x86 implementation. For the same reason, the underlying hardware can be changed radically without affecting legacy x86 software: each new CPU design only requires a new version of the Code Morphing software to translate x86 instructions to the new CPU's native instruction set. For the initial Transmeta products, models TM3120 and TM5400, the hardware designers opted for minimal space and power. By eliminating roughly three quarters of the logic transistors that would be required for an all-hardware design of similar performance, the designers have likewise reduced power requirements and die size. However, future hardware designs can emphasize different factors and accordingly use different implementation techniques. Finally, the Code Morphing software which resides in standard Flash ROMs itself offers opportunities to improve performance without altering the underlying hardware.

V. CODE MORPHING SOFTWARE

The Code Morphing software is fundamentally a dynamic translation system, a program that compiles instructions for one instruction set architecture (in this case, he x86 target ISA) into instructions for another ISA (the VLIW host ISA). The Code Morphing software resides in a ROM and is the first program to start executing when the processor boots. The Code Morphing Software supports ISA, and is the only thing x86 code sees; the only program written directly for the VLIW engine is the Code Morphing software itself. Figure1.shows the relationship between x86 codes, the Code Morphing software, and a Crusoe processor. Because the Code Morphing software insulates x86 programs—including a PC's BIOS and operating system—from the hardware engine's native instruction set, that native instruction set can be changed arbitrarily without affecting any x86 software at all. The only program that needs to be ported is the Code Morphing software itself, and that work is done once for each architectural change, by Transmeta. The feasibility of this concept has already been demonstrated: the native ISA of the model TM5400 is an enhancement (neither forward nor backward compatible) of the model TM3120's ISA and therefore runs a different version of Code Morphing software. The processors are different because they are aimed at different segments of the mobile market: the model TM3120 is aimed at Internet appliances and ultra-light mobile PCs, while the model TM5400 supports high performance, full-featured 3-4lb. mobile PCs. Coincidentally, hiding the chip's ISA behind a software layer also avoids a problem that has in the past hampered the acceptance of VLIW machines. A traditional VLIW exposes details of the processor pipeline to the compiler; hence any change to that pipeline would require all existing binaries to be recompiled to make them run on the new hardware. Note that even traditional x86 processors suffer from a related problem: while old applications will run correctly on a new processor, they usually need to be recompiled to take full advantage of the new processor implementation. This is not a problem on Crusoe processors, since in effect, the Code Morphing software always transparently "recompiles" and optimizes the x86 code it is running. The flexibility of the software-translation approach comes at a price: the processor has to dedicate some of its cycles to running the Code Morphing software, cycles that a conventional x86 processor could use to execute application code. To deliver good practical system performance, Transmeta has carefully designed the Code Morphing software for maximum efficiency and low overhead.

A. Decoding and Scheduling

Conventional x86 superscalar processors fetch x86 binary instructions from memory and decode them into micro-operations, which are then reordered by out of-order dispatch hardware and fed to the functional units for parallel execution. In contrast (besides being a software rather than a hardware solution), Code Morphing can translate an entire group of x86 instructions at once, creating a translation, whereas a superscalar x86 translates single instructions in isolation. Moreover, while a traditional x86 translates each x86 instruction every time it is executed, Transmeta's software translates instructions once, saving the resulting translation in a translation cache. The next time the (now translated) x86 code is executed; the system skips the translation step and directly executes the existing optimized translation. Implementing the translation step in software as opposed to hardware opens up new opportunities and challenges. Since an out-of-order processor has to translate and schedule instructions every time they execute, it must do so very quickly. This seriously limits the kinds of transformations it can perform. The Code Morphing approach, on the other hand, can amortize the cost of translation over many executions, allowing it to use much more sophisticated translation and scheduling algorithms. Likewise, the amount of power consumed for the translation process is amortized, as opposed to having to pay it on every execution. Finally, the translation software can optimize the generated code and potentially reduce the number of instructions executed in a translation. In other words, Code Morphing can speed up execution while at the same time reducing power!

B. Caching

The translation cache, along with the Code Morphing code, resides in a separate memory space that is inaccessible to x86 code .(For better performance, the Code Morphing software copies itself from ROM to DRAM at initialization time.) The size of this memory space can be set at boot time, or the operating system can make the size adjustable. As with all caching, the Code Morphing software's technique of reusing translations takes advantage of "locality of reference". Specifically, the translation system exploits the high repeat rates (the number of times a translated block is executed on average) seen in real-life applications. After a block has been translated once, repeated execution "hits" in the translation cache and the hardware can then execute the optimized translation at full speed. Some benchmark programs attempt to exercise a large set of features in a small amount of time, with little repetition—a pattern that differs significantly from normal usage. (When was the last time you used every other feature of Microsoft Word exactly once, over a period of a minute?) The overhead of Code Morphing translation is obviously more evident in those benchmarks. Furthermore, as an application executes, Code Morphing "learns" more about the program and improves it so it will execute faster and faster. Today's benchmarks have not been written with a processor in mind that gets faster over time, and may "charge" Code Morphing for the learning phase without waiting for the payback. As a result, some benchmarks do not accurately predict the performance of Crusoe processors.

C. Filtering

It is well known that in typical applications, a very small fraction of the applications code (often less than 10%, sometimes as little as 1%) accounts for more than 95% of execution time. Therefore, the translation system needs to choose carefully how much effort to spend on translating and optimizing a given piece of x86 code .The Code Morphing software includes in its arsenal a wide choice of execution modes for x86 code, ranging from interpretation (which has no translation overhead at all, but executes x86 code more slowly), through translation using very simple-minded code generation, all the way to highly optimized code (which takes longest to generate, but which runs fastest once translated).

D. Prediction and Path Selection

One of the many ways in which the Code Morphing software can gather feedback about the x86 programs is to instrument translations: the translator adds code whose sole purpose is to collect information such as block execution frequencies, or branch history. This data can be used later to decide when and what to optimize and translate. For example, if a given conditional x86 branch is highly biased (e.g. usually taken), the system can likewise bias its optimizations to favour the most frequently taken path. Alternatively, for more balanced branches (taken as often as not, for example), the translator can decide to speculatively execute code from both paths and select the correct result later. It would be extremely difficult to make similar decisions in a traditional hardware only x86 implementation .Current Intel and AMD x86 processors convert x86 instructions into RISC-like micro-ops that are simpler and easier to handle in a superscalar micro architecture. (In contrast, Cyrix and Centaur cores execute x86 instructions directly) The micro-op translation adds at least one pipeline stage and requires the decoder to call a microcode routine to translate some of the most complex x86 instructions. Implementing the equivalent of that fronted translation in software saves Transmeta a great deal of control logic and simplifies the design of its chips. It also allows Transmeta to patch some bugs in software. (The engineers fixed a timing problem in the TM5400 in this manner.) Some x86 chips, such as Pentium III, allow some patches to microcode, but these patches are very limited in comparison. Transmeta's software translation is a little more like the Motorola 68K emulation built into PowerPC-based Macs since 1994. What's new about

Transmeta's approach is that translation isn't merely an alternative to native execution—it's the whole strategy. Crusoe does for microprocessors what Java does for software: it interposes an abstraction layer that hides internal details from the outside world. Just as a Java programmer can write code without needing any knowledge about the underlying operating system or CPU, x86 programmers can continue writing software without needing any knowledge about a Crusoe system's VLIW architecture or code morphing software.

VI. LONGRUN POWER MANAGEMENT

Although the Code Morphing software's primary responsibility is ensuring x86 compatibility, it also provides interfaces to capabilities available only in Crusoe processor models. Longrun power management is one example—a facility in the TM5400 model that can further minimize that processor's already low power consumption. In a mobile setting, most conventional x86 CPUs regulate their power consumption by rapidly alternating between running the processor at full speed and (in effect) turning the processor off. Different performance levels can be obtained by varying the on/off ratio (the "duty cycle"). However, with this approach, the processor may be shut off just when a time-critical application needs it. The result may be glitches, such as dropped frames during movie playback, that are perceptible (and annoying) to a user. In contrast, the TM5400 can adjust its power consumption without turning itself off—instead, it can adjust its clock frequency on the fly. It does so extremely quickly, and without requiring an operating system reboot or having to go through a slow sequence of suspending to and restarting from RAM. As a result, software can continuously monitor the demands on the processor and dynamically pick just the right clock speed (and hence power consumption) needed to run the application—no more and no less. Since the switching happens so quickly, it is not noticeable to the user.

A. Longrun extends Battery Life

The TM5400's Longrun feature is one of the most innovative technologies introduced by Transmeta. To our knowledge, no other microprocessor can conserve power by scaling its voltage and clock frequency in response to the variable demands of software. Longrun can scale the CPU's voltage in as many as 32 steps, though in practice 5–7 steps are sufficient to achieve most of the benefits, according to Transmeta5 engineers. There are individually controllable, codependent ranges for voltage and frequency. In the current version of the TM5400, voltage can vary from 1.1 V to 1.6 V, and frequency can vary from 200 MHz to 700 MHz in increments of 33 MHz. Transmeta's software controls the scaling through a five-pin interface that adjusts an off-chip voltage regulator. When the Longrun software detects a change in the CPU load, it signals the chip to adjust the voltage and frequency up or down. If the CPU needs to handle a heavier load, Longrun tells the chip to start ramping up its voltage .When the voltage stabilizes at the higher level, the chip scales up its clock frequency. If the Longrun software determines that the CPU can save power by running more slowly, the chip starts scaling down its frequency. When the phase-lock loop (PLL) locks onto the lower clock rate, Longrun reduces the voltage. By always keeping the clock frequency within the limits required by the voltage, Longrun avoids any clock skewing or other undesirable effects. Longrun never needs more than one frequency step to reach a different target. To scale from 600 to 700 MHz, for instance, Longrun doesn't have to take three 33-MHz steps. Instead, it raises the voltage to 1.6 V in multiple steps, and then boosts the frequency to 700 MHz in one big jump. This avoids the latencies of resetting the frequency multiple times .One concern is that Longrun might not react quickly enough to accommodate the fast changing demands of some programs. When the computer is playing MPEG compressed video, for example, a transition from a relatively static frame to an action-filled frame might overwhelm a CPU that's comfortably loafing at a low clock speed. MPEG compression works by saving the differences between frames, and the frames are only 1/30 of a second apart. The CPU load would vastly increase after a sudden transition from a speech to a car chase. But Crusoe Longrun software can detect a change in the CPU load in about half a microsecond, and Longrun can scale the voltage up or down in less than 20 microseconds per step. The worst-case scenario of a full swing from 1.1 V to 1.6 V and from 200 to 700 MHz takes only 280 microseconds. Furthermore, the CPU doesn't stall during the swing. The processor keeps executing instructions, stalling only while the PLL relocks onto the new frequency. That doesn't take longer than 20 microseconds in the worst case, and Transmeta's engineers say they've never observed a relock taking longer than 10 microseconds. Longrun isn't the only reason that Crusoe processors appear to consume much less power than comparable x86 chips. The TM3120 doesn't have Longrun, yet its power consumption is impressive too. The simplicity of Transmeta's VLIW architecture is evidently a larger factor. Longrun is a genuine innovation that gives Crusoe an extra edge. Finally, the Code Morphing software can also adjust the Crusoe processor's voltage on the fly (since at a lower operating frequency, a lower voltage can be used). Because power varies linearly with clock speed and by the square of the voltage, adjusting both can produce cubic reductions in power consumption whereas conventional CPU scan adjust power only linearly. For example, assume an application program only requires 90% of the processor's speed. On a conventional processor, throttling back the processor speed by 10% cuts power by 10%, whereas under the same conditions, Longrun power management can reduce power by almost 30%—a noticeable advantage.

VII. CRUSOE PROCESSOR HARDWARE ARCHITECTURE

A. Crusoe Processor Architecture

The Crusoe microprocessor is available in the market in the following versions: TM3120, TM3200, TM5400 and TM5600. The basic architecture of all the above models is same except for some minor changes since various models have been introduced for different segments of the mobile computing market. The following architectural description has taken Crusoe TM5400 as reference. The Crusoe Processor incorporates integer and floating point execution units, separate instruction and data caches, a level-2 write-back cache, memory management unit, and multimedia instructions. In addition to these traditional processor features, the device integrates a DDR SDRAM memory controller, SDR SDRAM memory controller, PCI bus controller and serial ROM interface controller. These additional units are usually part of the core system logic that surrounds the microprocessor. The VLIW processor, in combination with Code Morphing software and the additional system core logic units, allow the Crusoe Processor to provide a highly integrated, ultra-low power, high performance platform solution for the x86 mobile market.

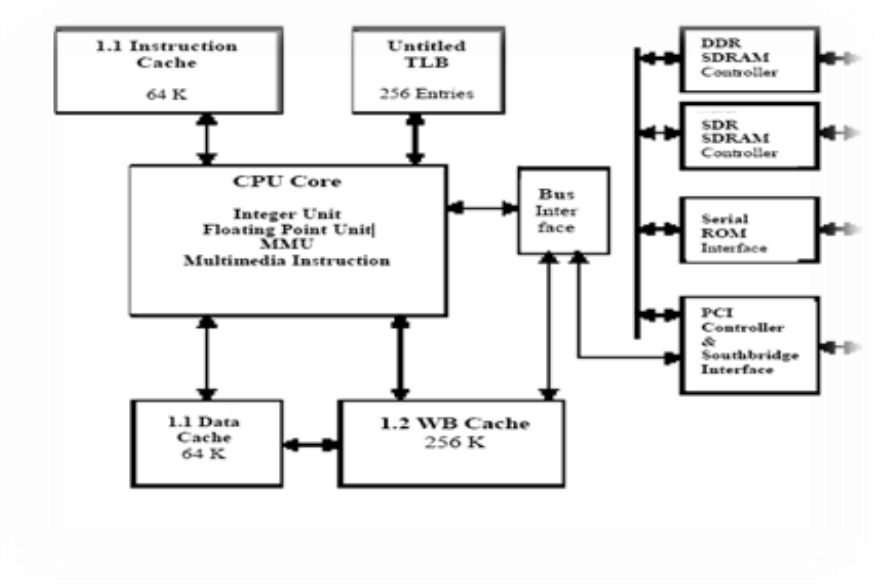

Fig: Crusoe processor Architecture-Model TM5400

B. PROCESSOR CORE

The Crusoe Processor core architecture is relatively simple by conventional standards. It is based on a Very Long Instruction Word (VLIW) 128-bit instruction set. Within this VLIW architecture, the control logic of the processor is kept very simple and software is used to control the scheduling of instructions. This allows a simplified and very straightforward hardware implementation with an in-order 7- stage integer pipeline and a 10-stage floating point pipeline. By streamlining the processor hardware and reducing the control logic transistor count, the performance-to-power consumption ratio can be greatly improved over traditional x86 architectures. The Crusoe Processor includes a 8-way set-associative Level 1 (L1) instruction cache, and a 16-way set associative L1 data cache. It also includes an integrated Level 2 (L2) write-back cache for improved effective memory bandwidth and enhanced performance. This cache architecture assures maximum internal memory bandwidth for performance intensive mobile applications, while maintaining the same low-power implementation that provides a superior performance-to-power consumption ratio relative to previous x86 implementations.

C. INTEGRATED DDR SDRAM MEMORY CONTROLLER

The DDR SDRAM interface is the highest performance memory interface available on the Crusoe Processor. The DDR SDRAM controller supports only Double Data Rate (DDR) SDRAM and transfers data at a rate that is twice the clock frequency of the interface. This feature is absent in the Crusoe processor model TM 3200. The DDR SDRAM controller supports up to four banks, the equivalent of two Dual line Memory Modules (DIMM's), of DDR SDRAM using a 64-bit wide inter-face. The DDR SDRAM memory can be populated with 64M-bit, 128M-bit, or 256M-bit devices. The frequency setting for the DDR SDRAM interface is initialized during the power-on boot sequence.

D. INTEGRATED SDR SDRAM MEMORY CONTROLLER

The SDR SDRAM memory controller supports up to four banks, equivalent to two Small Outline Dual In-line Memory Modules (SO-DIMMS), of Single Data Rate (SDR) SDRAM that can be configured as 64-bit or 72-bit SO-DIMM's. This SODIMM's can be populated with 64M-bit, 128M-bit or 256M-bit devices. All SODIMM's must use the same frequency SDRAM's, but there are no restrictions on mixing different SO- DIMM configurations into each SO-DIMM slot. The frequency setting for the SDR SDRAM interface is initialized during the power-on boot sequence.

E. INTEGRATED PCI CONTROLLER

The Crusoe Processor includes a PCI bus controller that is PCI 2.1 compliant. The PCI bus is 32 bits wide, operates at 33 MHz, and is compatible with 3.3V. Signal levels. It is not 5V tolerant, however. The PCI controller on provides a PCI host bridge, the PCI bus arbiter, and a DMA controller.

F. SERIAL ROM INTERFACE

The Crusoe Processor serial ROM interface is a five-pin interface used to read data from a serial flash ROM. The flash ROM is 1M-byte in size and provides non-volatile storage for the Code Morphing software. During the boot process, the Code Morphing code is copied from the ROM to the Code Morphing memory space in SDRAM.

VIII. CONCLUSIONS

In 1995, Transmeta set out to expand the reach of microprocessors into new markets by dramatically changing the way microprocessors are designed. The initial market is mobile computing, in which complex power-hungry processors have forced users to give up either battery running time or performance. The Crusoe processor solutions have been designed for lightweight (two to four pound) mobile computers and Internet access devices such as handhelds and web. They can give these devices PC capabilities and unplugged running times of up today. To design the Crusoe processor chips, the Transmeta engineers did not resort to exotic fabrication processes. Instead they rethought the fundamentals of microprocessor design. Rather than "throwing hardware" at design problems, they chose an innovative approach that employs a unique combination of hardware and software. Using software to decompose complex instructions into simple atoms and to schedule and optimize the atoms for parallel execution saves millions of logic transistors and cuts power consumption on the order of 60–70% over conventional approaches—while at the same time enabling aggressive code optimization techniques that are simply not feasible in traditional x86 implementations. Transmeta's Code Morphing software and fast VLIW hardware, working together, achieve low power consumption without sacrificing high performance for real-world applications. Although the model TM3120 and model TM5400 are impressive first efforts, the significance of the Transmeta approach to microprocessor design is likely to become more apparent over the next several years. The technology is young and offers more freedom to innovate (both hardware and software) than conventional hardware-only designs. Nor is the approach limited to low-power designs or to x86-compatible processors. Freed to render their ideas in a combination of hardware and software, and to evolve hardware without breaking legacy code, Transmeta microprocessor designers may produce one surprise after another in the coming years.

ACKNOWLEDGMENT

The Author would like to thank to Dr. P. P. Karde and prof. P. L. Ramteke, department of computer science and information technology, HVPM COET, Amravati, India.

REFERENCES

[1] J. C. Dehnert , "The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges," *IEEE/ACM Symp. Code Generation and Optimization*, San Francisco,CA, Mar 2003 .
[2] J.C. Dehnert, "Using Performance Counters for Runtime Temperature Sensing" in High-Performance Processors , July 2003.
[3] James C. Dehnertetal, "Transmeta's Magic Show ," *IEEE* SPECTRUM, May 2000.
[4] John P. Banning, "The Transmeta Code Morping, Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges",March2003
[5] A. Klaiber, "The Technology Behind Crusoe Processors",July 2000.
[6] Linley P. Gwenna, "LLVA: A Low-level Virtual Instruction Set Architecture", April 2002.
[7] Rob Hughes, "The Crusoe Processor ",January 2000.

[8] Alexander C. Klaibe, "Transmeta crusoe", January 2000.
[9] Sun Microsystems, "The Java Hotspot Performance Engine Architecture," April 1999.
[10] Malcolm J. Wing and Godfrey P. D'Souza, "Gated store buffer for an advanced microprocessor," US Patent 6,011,908, Jan. 2000.
[11] Byung-Sun Yang et al, "LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation," Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Oct. 1999.
[12] Frank Yellin and Tim Lindholm, The Java Virtual Machine Specification, Addison-Wesley, 1996.
[13] Cindy Zheng and Carol Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," IEEE Computer 33 (3), March 2000, pp.
[14] Edmund J. Kelly, Robert F. Cmelik, and Malcolm J. Wing, "Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed," US Patent 5,832,205, Nov. 1998.
[15] Alexander Klaiber, "The Technology Behind the Crusoe Processors," Jan. 2000.

