Optimizing Intel EPIC/Itanium2 Architecture for Forth

Jamel Tayeb*, Smail Niar**
*Intel Corporation, Portland, Oregon (USA)
**LAMIH ROI, University of Valenciennes, (France)
Jamel.Tayeb@intel.com, Smail.Niar@univ-valenciennes.fr

Abstract

Forth is a stack machine that represents a good match for the register stack of the Explicit Parallel Instruction Computer (EPIC) architecture. In this paper we will introduce a new calling mechanism using the register stack to implement a Forth system more efficiently. Based upon our performance measurements, we will show that the new calling mechanism is a promising technique to improve the performance of stack-based interpretative languages such as Forth. The limitation in EPIC's Register Stack Engine makes the need for hardware support to improve performance and possibly close the efficiency gap with specialized stack processors. We will define also an adjustment to Itanium 2 processor's instruction set to accommodate the new calling mechanism and present a conservative architectural implementation over the current Itanium 2 processor's pipeline.

1. Introduction

1.1. Background

Virtual machines are an effective ways to take advantage of the increasing chip and system-level parallelism – introduced *via* technologies such as simultaneous multi-threading [1], multi-core designs [2] and systems-on-a-chip (networks) [3]. The performance of a virtual machine depends on its implementation and its interaction with the underlying processing architecture [4].

Just in Time [5] and Adaptive Dynamic Compilation [6] techniques were developed to provide performance gain over pure interpretation. In practice just in time and adaptive dynamic compilation suffer some limitations. In particular, it is difficult to explore a large set of optimizations in a limited period of time. This issue makes most just in time compilers to narrow down the field and the scope of their optimizations. They also require additional memory, which may be impractical in an embedded environment.

1.2. Project aim

The aim of our work is to close as much as possible the theoretical efficiency gap that exists between EPIC (Explicit Parallel Instruction Computer) [7] and stack processor architectures while running Forth applications [8]. To do so, we are comparing the Itanium 2 processor's register stack to existing stack processors' architectures using Forth as their assembly language (in section 6). Forth is used in the scope of this study because it is a simple stack machine [9]. This makes it well suited as a proxy for more sophisticated stack machines such as .NET (The MSIL evaluation stack). In addition, Forth's key intrinsic advantages are:

- A low memory footprint;
- A high execution speed;
- The ability to interactively expand its dictionaries while developing applications.

1.3. Why using EPIC?

Itanium processors are today the only commercial chips to implement the EPIC architecture. This processor family is specifically targeting the enterprise server and high-performance computing cluster segments. With 410 million transistors required to implement the EPIC architecture in the Itanium 2 processor (9MB on-chip cache memory), one can argue that IPF doesn't seem to be well suited for mid or low range, or even embedded applications. However, the EPIC architecture is not reserved to the high-end servers and offers enough flexibility – I.e. the execution window width of the machine – to adapt it to specific needs. It is also interesting to notice that the Itanium 2 processor core uses less than 30 million transistors to implement the processor's logic (where a modern x86, out-of-order execution engine's implementation requires 40+ million transistors). The reminder of the transistors budget is essentially dedicated to build the huge on-chip cache memory (Level 3 essentially). It is therefore realistic to consider the design of a low-end processor based on EPIC architecture and having a limited amount of on-chip cache memory (128KB L2 and/or 1MB L3). In consequence of that:

- EPIC architecture, with its large register file and its simple and in-order core makes it well suited to host a stack machine, such as Forth,
- Itanium 2 processor is a good development vehicle and the best performance proxy available for our initial study.

1.4. Plan

We first introduce in section 2 a new Stack Indexed Register (SIR) based on Itanium 2 processor's register stack to implement a purely software virtual machine, running Forth. Based upon our performance projections (summarized in section 5), we demonstrate that the proposed mechanism is a promising technique to improve the performance of stack-based interpretative virtual machine. But limitation in EPIC's register stack engine makes the need for a hardware support to reach optimal

performance and close as much as possible the theoretical efficiency gap with stack processors (detailed in section 6.1 – related projects). In section 3, we define an addition to Itanium 2 processor's instruction set to accommodate the SIR. In section 4, we describe a conservative architectural implementation of the extended instruction set. We summarize our experimental results in section 5 and present our conclusions in section 7.

2. The New Calling Convention

Our reference Forth virtual machine is threaded and uses in-memory stacks. Parameter passing is done through the stack, and an optimizing compiler (Microsoft Visual C++ 2005 for Itanium) is used to generate the binary of words defined in the X3.215-1994 ANS standard [10]. Assembly coding is done using ias, the Intel EPIC assembler.

First, to present the use by compilers of the Itanium 2 processor register stack, let's examine a function call using the address interpreter's principal statement – performing NEXT: `(pf->internals.ip->cfa)(pf);`

The translation of this statement by the compiler in EPIC assembly language is given in Table 1.

Table 1 - Translation in the EPIC assembly language of (pf->internals.ip->cfa)(pf);

```
        { .mii
1          alloc r35=2,3,1,0
2          mov r34=b0
3          adds r31=528, r32
        } … { .mmb
4          mov r36=gp
5          mov r37=r32
6          nop.b 0;;
        } { .mmi
7          ld8 r30=[r31];;
8          ld8 r29=[r30]
9          nop.i 0;;
        } { .mmi
10         ld8 r28=[r29], 8;;
11         ld8 gp=[r29]
12         mov b6=r28
        } { .mmb
13         nop.m 0
14         nop.m 0
15         br.call.dptk.many b0=b6;;
        }
```

The function call itself is clear enough – the target address is stored in the b6 branch register (instruction 12 and 15 for the actual branching). The key operation for the function call mechanism is the alloc instruction (instruction 1). It allocates a new stack frame to the register stack. By specifying the number of input, output, local – and rotating registers – required at the beginning of the procedure to the register stack engine, the caller sets the arguments for the callee. Note that the alloc instruction can be used anywhere in a program and as many times as needed. Any consecutive instruction to the alloc will immediately see the renamed registers. Here, the pf pointer is directly and always available in the general-purpose register r32 and can be used right away to compute the interpreting pointer (ip) address. This mechanism is well suited to support object-oriented languages which tend to be dominated by calls to low instruction-count functions.

Even if the register stack engine provides an efficient way to pass arguments back and forth all along the call stack, our reference Forth implementation still has to manage its in-memory stacks. In consequence, we introduce our SIR to allow the compiler to keep the entire – or partial – Forth stack in the register stack.

Let's consider the simple + word, summing two numbers on the stack. The reference code in C is:

```
void CORE_PLUS(PFORTH pf) {
   int3264 n1, n2;
   POP(n2); POP(n1); PUSH(n1 + n2);
}
```

In the proposed mechanism, a sub-set of the Itanium 2 processor register file (the stacked registers) is recycled as an in-register data and floating-point stack. The return stack can either be mapped into the branch registers of the processor or in the general purpose register file. The major technical difficulty consists here in maintaining the stack size in the Forth interpreter – forcing the Forth compiler to compute the words' arity – and using self-modifying code to adjust the alloc instruction's arguments accordingly after each return from the primitives. This coding technique leads to a functional Forth engine but suffers some limitations. The alloc instruction cannot allocate a stack frame larger than 96 registers. Yet, if needed, additional stack elements are spilled / filled by the register stack engine into the backing store memory, with a performance overhead. A secondary limitation of using the stacked registers as in-register stack is that it may limit the use of the software pipelining (a key performance technique for Itanium 2 processor [11]) within the Forth words by the compiler.

As soon as the stack size limitation is satisfied, we can support the Forth virtual machine in a much more efficient way. It is noticeable that the performance benefit of the SIR is increasing proportionally with the amount of stack handling primitives used by the code. The entire execution of + can now be scheduled for only two processor cycles as shown in the next listing. Note that this code was hand-written and differs therefore from the compiler generated assembler listed in table 1 – not showing the bundles explicitly.

```
.global SIR_CORE_PLUS
.type SIR_CORE_PLUS, @function
.proc SIR_CORE_PLUS
pfs = r34
SIR_CORE_PLUS:
;alloc placeholder
alloc pfs = 2, 1, 1, 0 ;default arity
add out0 = in0, in1
mov ar.pfs = pfs
br.ret.sptk.many b0
```

```
.endp
```

Table 2 compares the principal characteristics of both implementations of +. A bundle is a group of three instructions. A stop bit is introducing a serialization in the instruction stream.

Table 2 - Characteristics of the two versions of +.

Features	Reference Implementation	Proposed optimized implementation
I/FP registers	9/0	2/0
Bundles	14	2
Nops	5	3
Stop bits	10	1
Branches	6	1
Loads	6	0
Stores	1	0

The second advantage of the SIR is that we can still entirely rely upon the register stack engine to trap and process stack overflow exceptions in exchange of a performance penalty. When such condition happens during the execution of the alloc instruction – I.e. insufficient registers are available to allocate the desired stack frame – the processor stalls until enough dirty registers are written to the backing store area (these stall cycles can be monitored for optimization purpose through the BE_RSE_BUBBLE-ALL performance counter [12]).

Alas, EPIC doesn't provide the same register-passing mechanism for floating-point arguments. This lack makes necessary to manage the floating-point register file explicitly to implement the SIR, making the compiler more complex and asymmetrical for integer and floating-point stack handling. But having a large on-chip floating-point register file (128 registers) and the associated computing resources (2 floating-point execution units capable of vector operations – up to 4 FLOP per cycle) still provides a considerable performance advantage over stack processors for floating-point intensive codes.

By using Itanium 2 processor's register files as in-register stacks, it is possible to eliminate:

- The need for the pop / push primitives, which are embedded into the EPIC Register Stack Engine – at least for the integer operations;
- The multiple clock-cycle floating-point load instructions required for passing the argument *via* the in-memory floating-point stack (for reference: 13 cycles for L3 hit, 6 cycles for L2 hit and 1 cycle for L1 hit – integer data only in L1D);
- The energy consumption and power dissipation associated with the suppressed loads / stores from / to cache / memory.

With the Itanium 2 processor, up to 96 general purpose registers can be used to implement the Forth data stack and 96 floating-point registers to implement the optional floating-point stack. In our implementation, the data is mapped as follows:

- Data stack: r32-r127,
- floating-point stack: f32-f127,
- And Return stack: b6-b7 (can be mapped into the integer register file).

Our software implementation of the SIR has an additional drawback when it is used in conjunction of the standard calling mechanism. It requires extra code and processor cycles to ensure the register spilling / filling when switching between calling conventions. This is currently mitigating the performance gains on applicative benchmarks[1] as only a limited set of Forth primitives are implemented using the SIR.

3. Enhancing the Itanium 2 processor instruction set to Support SIR

To overcome the software implementation's limitation and to generalize the SIR's usage between the integer and floating-point register files, we propose a global hardware indexed access to the register files. We assume the following notations: gr[reg] or gr[imm] and fr[reg] or fr[imm] where:

- gr is the general-purpose register file and fr is the floating-point register file;
- reg is the register that holds the index into the register file;
- imm is the index value into the register file.

Here after, we will describe only the integer case as the floating-point case can be directly derived. Let's assume the following convention for the stack index registers to recode the Forth virtual machine with the modified instruction set:

- Index to Data Stack TOS (gr_tos) = r2;
- Index to Data Stack level 1 (gr_l1) = r3;
- Index to Data Stack level 2 (gr_l2) = r14;
- Index to Forth Data Stack level 3 (gr_l3) = r15.

These registers were selected to simplify the co-existence of SIR with the standard calling convention as they are unused and unsaved during standard calls. However, any register (lower than r32 and fr32 could be used as indexes – at the exception of the read-only r0, r1, f0 and f1 registers).

In consequence, coding + no longer requires the register stack engine and the integer data stack is managed in the same way as the floating-point stack. The required comparison and the extra additions needed to detect the stack underflow situation and to maintain the stack pointers up-to-date are not penalizing because of the underlying VLIW nature of the EPIC architecture. This allows us to reuse the otherwise empty (nop) bundle slots to perform the required operations. It is also interesting to notice that the predicate registers (p6 and p0) allow expressing the test and the branch instruction if true in a very compact way. With our proposed instruction set addition, the code for +, embedding the stack management can still be scheduled for two processor cycles and is listed below:

```
.global SIR_CORE_PLUS
```

[1] This overhead can be removed by coding the entire Forth virtual machine in assembler using our SIR rather than using also a C++ compiler – a task which is out of the scope of this study.

```
  .type SIR_CORE_PLUS, @function
  .proc SIR_CORE_PLUS
  SIR_CORE_PLUS:
  cmp4.lt.unc p6, p0 = 32, gr_tos
  (p6) br.cond.dptk.many
@underflow_exception;;
  add gr[gr_l1] = gr[gr_tos],
gr[gr_l1];;
  mov gr_tos = gr_l1;;
  add gr_l1 = -1, gr_tos
  add gr_l2 = -2, gr_tos
  add gr_l3 = -3, gr_tos
  br.ret.sptk.many b0;;
  .endp
```

4. A Conservative Implementation

By limiting further the number of registers used as our in-register stacks to 64 we can propose a conservative architectural implementation of the SIR that would not require an instruction set modification. The new simplified logical view of the register files and the in-register stacks is shown in Figure 1. It is the compiler's responsibility to enforce the segregation between the in-register stacks and the traditional register file.

We first define a new indexed capability for the higher 64 registers identified *via* the CPUID instruction. An additional bit in the status register indicates if the functionality is enabled. If not, the additional Register Alias Table (RAT) required by our implementation – described later – is bypassed and no recompilation of existing code is required to run as-is. A compiler willing to use the SIR has to check if the functionality is available – on the target system – and to activate at runtime the in-register stacks by updating the status register.

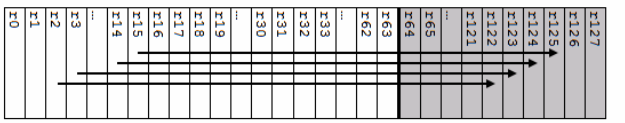

Figure 1 - Snapshoot showing the logical view of the integer register file. In grey the recycled register files subset as in-register stacks. Arrows represent the indexing.

When the in-register stacks are active, the EXP (Template decode, Expand and Disperse) stage of the core pipeline has to check, per instruction, if the MSB of a source register is set (noted MSB Detect in Figure 2). If not, then the normal execution of the instruction takes place. If the MSB is set for at least one register, then the additional RAT checks if the target register is to be modified by an instruction currently executed. To track the status (ready / not ready) of the target registers, the RAT uses a 64 x 1 bit vector. If the corresponding ready bit is set, then the RAT feeds into the REN stage the new register address (using a multiplexer and a latch - one per indexed register – holding the 6 bits of the real register address in the register file (noted Index Register Cache in Figure 2). If the register is marked as not ready in the RAT, then a serialization must take place, and a pipeline stall happens. Once the target register is ready, its value if forwarded into its corresponding latch of the RAT, which updates the register's status bit. The stalled instruction's execution can therefore be resumed.

Our simplified implementation allows indexed access to only 64 registers in the integer and floating-point register files. It also requires 1 bit in the CPUID, 1 bit in the status register and an MSB bit-set detection during the early stages of the instruction decoding. It also requires a 64-entry RAT using 64 x 6-bit latches and multiplexers, plus 64 x 1 status bit vector; and adds an extra execution cycle to the main pipeline. In return, it provides the following advantages:

- Implements the required integer and floating-point in-register stacks, under the compiler's control (limited to 64-integer and 64 floating-point entries);
- It is possible to implement with the actual Itanium processor pipeline;
- It is totally compatible with existing software;
- It also allows:
  - The suppression of the loads / stores associated with stack operations (hence ensuring performance gains over C code);
  - The substantial reduction of the chip's power consumption when executing stack handling routines, a dominant in Forth applications and virtual machines in general.

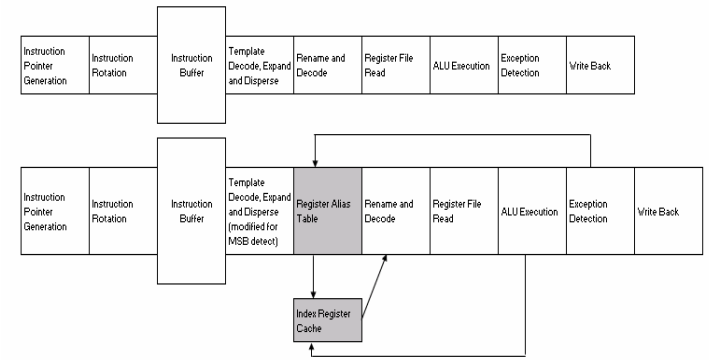

Figure 2 - the current – simplified – main pipeline (top) and the modified one (bottom). Additional structures are marked in grey.

5. Experimental Results

In this section, we present the results of our experimental software implementation of the SIR. We have benchmarked 11 major stacks handling Forth words along with the integer and floating-point additions. Each of these words was recoded using the software implementation of the SIR. Performance was measured by averaging the number of processor cycles required to execute a billion occurrences of each word (measured by using the processor's interval time counter application register – ar.itc). Our performance measurements demonstrate that it is appropriate to consider the EPIC register files as a set of in-register stacks to run a

virtual machine, and particularly a Forth virtual machine. We measured speed-ups ranging from a low 1.95 to a high 15.6 (Table 3).

Although the simplified architectural implementation described in section 4 is not realized, our performance data provides a realistic projection of the performance that could be reached by using the hardware implementation of the SIR. Because Forth routines and virtual machines in general are heavily using stack manipulations, the measurable performance gains in these synthetic benchmarks are likely to be directly translatable into application-level performance gains.

Table 3 – Summary of performance measurements.

Word implementation	CPU Cycles	Speed-up
core_plus (+)	29.25	-
sir_core_plus (+)	15.00	1.95
core_two_dup	48.00	-
sir_core_two_dup	5.00	9.60
core_two_over	78.00	-
sir_core_two_over	5.00	15.60
core_two_swap	62.00	-
sir_core_two_swap	6.00	10.33
core_dup	28.00	-
sir_core_dup	5.00	5.60
core_over	41.00	-
sir_core_over	6.00	6.83
core_rot	48.00	-
sir_core_rot	5.00	9.60
core_swap	33.00	-
sir_core_swap	5.00	6.60
floating_f_plus (f+)	44.00	-
sir_floating_f_plus (f+)	13.25	3.32
floating_fdup	43.00	-
sir_floating_fdup	8.00	5.38
floating_fover	64.00	-
sir_floating_fover	14.00	4.57
floating_frot	66.00	-
sir_floating_frot	7.00	9.43
floating_fswap	51.00	-
sir_floating_fswap	7.00	7.29

6. Related projects

6.1. Specialized processors

The Forth community has explored the potential of designing custom microcontrollers to efficiently run the Forth language. Although each custom design has its own unique objectives and approach to the problem statement, three significant common characteristics to the most successful designs can be noted:

- The integration of at least two distinct memories into the processor. These memories are used as the Forth data and return stacks [13,14,15,16]. In principle, the number of stacks is not limited, and each stack may have a very specific role, as in the Stack Frame Computer [13].
- The presence of a few dedicated registers for managing the stacks. The bare minimum is the Top of the Stack (TOS) or stack pointer: one for the data and one for the return stack. To permit quick access to data buried deep in the stacks, a set of additional registers may be implemented. By writing a value into these registers, it is possible to generate the address of any stack level, as illustrated in the HS-RTX microcontrollers [14].
- The short latency of the instruction execution, which is often reduced to a single cycle. This allows the language's key primitives to be implemented efficiently. Multiple paths can be taken to reach this goal: a simple cache of the stack's top elements can be created in registers that feed directly into the ALU (e.g., Writable Instruction Set Computer [15]) or overlapped bus cycles can be combined (e.g., Minimum Instruction Set Computer and the Forth Reduced Instruction Set Computer [16]). The Forth Reduced Instruction Set Computer, for example, can read both the TOS and any of the first four stack elements (from the data and return stacks) within the same cycle, using dedicated and independent busses.

The open-source MicroCore project is one of the most recent implementations of a specialized microcontroller that uses the Forth language as its assembler. (It can also execute other languages, such as C) [17]. This microcontroller has an on-chip data and return stack, can directly implement 25 Forth primitives, and is capable of executing each instruction in a single clock-cycle.

Still, Forth is not the only stack-oriented language that encourages specific circuitry designs to achieve maximum performance. Java processors – such as the Sun Picojava and Imsys Cjips chips [18,19] – are also good examples of custom designs implementing a dedicated stack engine (the dribbler). The IBM zSeries Application Assist Processors (zAAPs) also provides a dedicated HW assist to asynchronously execute eligible Java code within the WebSphere JVM under the central processors' control [20].

6.2. General purpose processors

A parallel research path studies the use of general purpose processor's registers to perform stack caching. The caching technique can be used to statically and / or dynamically cache various stack levels [21,22,23]. Promising performance gains were demonstrated (up to x3.8 speedup – variable with the underlying processor architecture and code's nature) but these techniques also showed limitations when increasing the number of cached stack elements – over 3 – as the static and the dynamic caching techniques require to maintain multiple copies of the code based on the possible cache states. This last task is the interpreter or the compiler's responsibility. Stack caching, used in conjunction with code caching techniques, was used to limit code bloat [24].

The Philips TriMedia VLIW processor was used with a three stage software pipelined interpreter to achieve a peak

sustained performance of 6.27 cycles per instruction [25]. Interpretation is used by the authors to compress non-time-critical code, where time-critical-code is compiled to native code.

7. Conclusions

We presented an innovative use model for the Itanium 2 processor register files to improve Forth systems' performance running on EPIC architecture. Synthetic benchmarking shows an average 7x performance increase over the code generated by a state-of-the-art C/C++ compiler, using EPIC's standard calling convention (from 1.95x up to 15.6x).

Based upon our findings and coding experiments, we introduced an adjustment to the Itanium 2 processor instruction set offering indexed register file access, to ease Forth systems' implementation and increase its efficiency.

We then proposed an architectural implementation of a limited version of the adjustment – by restricting the size of the Forth integer and floating-point in-register stacks to 64 entries each –, making it conceivable to implement into the current Itanium 2 processor's pipeline. If realized, this adjustment should lead to a more efficient use of the register files to host a virtual machine's data and control stacks. By mapping the Forth stacks into the register files instead of the main memory, the load and store operations associated to the stack handling primitives would be suppressed, allowing performance gains associated to power savings.

8. Acknowledgment

The authors would like to thank Intel Corporation and particularly the Microprocessor Technology Labs (http://www.intel.com/technology/computing/mtl/) for the support given to this work. We also would like to thank the referees for their insightful comments that have improved this paper.

9. References

[1] D. Tullsen, S. Eggers, and H. Levy: "Simultaneous Multithreading: Maximizing On-Chip Parallelism", in Proceedings of the 22nd AISCA conference, June 1995.

[2] P. P. Gelsinger, Intel Corporation, Hillsboro, OR, USA: "Microprocessors for the New Millennium –Challenges, Opportunities and New Frontiers", in IEEE ISSC, 2001.

[3] L. Benini and G. De Micheli: "Networks on Chip: A New Paradigm for System on Chip Design", in Proceedings of the 2002 DATE conference 2002.

[4] J. Smith and R. Nair: "Virtual Machines: Versatile Platforms for Systems and Processes", Elsevier Science & Technology Books, May 2005.

[5] T. Shpeisman, G-Y. Lueh and A-R. Adl-Tabatabai, "Just-In-Time Java Compilation for the Itanium Processor", 11th PACT conference, 2002, p. 249.

[6] D. Bruening, T. Garnett and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization", Code Generation and Optimization, 2003, pp. 265-275.

[7] M. S. Schlansker and B. Ramakrishna Rau: "EPIC: Explicitly Parallel Instruction Computing", in IEEE Computer Society Press, Volume 33, Issue 2 (February 2000), pp 37-45.

[8] E. D. Rather, D. R. Colburn and C. H. Moore, "The Evolution of Forth", ACM SIGPLAN Notices, Volume 28, No. 3, March 1993.

[9] P. Koopman, Jr.: "Stack Computers: the new wave", Ellis Horwood (1989), republished on the World Wide Web by Mountain View Press.

[10] American National Standard for Information System, Technical Committee X3J14, "X3.215-1994: Programming Languages – Forth", 1994.

[11] S. Niar and J. Tayeb: « Programmation et Optimisation d'Applications pour les Processeurs Intel Itanium », Editions Eyrolles, January 2005.

[12] Intel Corporation, "Intel Itanium 2 Processors Reference Manual for Software Development and Optimization", Volumes 1, 2 and 3.

[13] R. D. Dixon, M. Calle, C. Longway, L. Peterson and R. Siferd: "The SF1 Real Time Computer" Proceedings of the IEEE National Aerospace and Electronics Conference, Dayton, OH, Vol. 1, pp. 60-64, May 1988.

[14] T. Hand: "The Harris RTX 2000 Microcontroller", Journal of Forth Application and Research, Vol. 6, No. 1, pp. 5-13, 1990; and the Interstil "Radiation Hardened Real Time Express™ HS-RTX2010RH Microcontroller Data Sheet.

[15] P. Koopman: "Writable Instruction Set Stack Oriented Computers: The WISC Concept", Journal of Forth Application and Research (Rochester Forth Conference Proceedings), vol. 5, no. 1, pp. 49-71, 1987.

[16] J. R. Hayes and S. C. Lee: "The Architecture of FRISC 3: A Summary", 1988 Rochester Forth Conference Proceedings, 1988, Institute for Applied Forth Reserch Inc.

[17] K. Schelisiek: "MicroCore: an Open-Source, Scalable, Dual-Stack, Hardware Processor Synthesisable VHDL for FPGAs", euroForth 2004.

[18] J. Michael O'Connor and Marc Tremblay, "PicoJava-i: The Java Virtual Machine in Hardware", Micro, IEEE, Volume 17, Issue 2, March-April 1997, pp. 45-53.

[19] Imsys Technologies AB, "IM1101C – the Cjip – Technical Reference Manual", www.imsys.se/documentation/manuals/tr-CjipTechref.pdf, 2004.

[20] IBM Redbook on zAAP: SG24-6386 (www.redbooks.ibm.com)

[21] A. Ertl and D. Gregg: "Stack Caching in Forth", in EuroForth 2005.

[22] A. Ertl:. "Stack Caching for Interpreters", in SIGPLAN '95 Conference on Programming Language Design and Implementation, pp. 315-327, 1995.

[23] K. Ogata, H. Komatsu and T. Nakatani: "Bytecode Fetch Optimization for a Java Interpreter", in ASPLOS 2002, pp. 58-67, 2002.

[24] P. Peng, G. Wu and G. Lueh: "Code Sharing among States for Stack-Caching Interpreter", in Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators, pp. 15-22, 2004.

[25] J. Hoogerbrugge, L. Augusteijn, J. Trum, R. van de Wiel: "A Code Compression System Based on Pipelined Interpreters. Software – Practice and Experience 29(11): 1005-1023, 1999.