An MSIL Hardware Evaluation Stack for EPIC

Jamel Tayeb
Intel Corporation, Portland, Oregon, USA
Jamel.tayeb@intel.com

Smail Niar
LAMIH ROI, University of Valenciennes, France
smail.niar@univ-valencienne.fr

Nasser Benameur
LAMIH ROI, University of Valenciennes, France
nasser.benameur@univ-alencienne.fr

Abstract

In this paper, we define a hardware Evaluation Stack for MSIL and we propose a conservative implementation over the EPIC architecture. This new hardware evaluation stack, named virtual stack, is based upon the EPIC architecture's register files. An additional register renaming logic, to offload the run-time type checking performed by the Common Language Runtime is also proposed. Finally, we introduce a mechanism to overcome the sequential nature of the evaluation stack allowing the code generator to better use the parallel instructions execution of instruction bundles. The virtual stack's final purpose is to simplify the implementation of fast one-pass JIT compiler.

***Keywords**: EPIC, MSIL, Hardware Virtual Stack.*

1 Introduction

The Itanium processor family's architecture – also known as EPIC for Explicit Parallel Instruction Computer – has been introduced to satisfy enterprise backend servers' and high performance computing clusters' performance requirements. In the same time, virtual machines and their associated languages and frameworks such as Java and .NET have become ubiquitous thanks to the great software development and deployment flexibility they provide. The association of both technologies provides a compelling hardware and software stack to fuel modern applications.

The aim of our work is to provide a hardware support for the Microsoft Intermediate Language (MSIL) evaluation stack based upon the EPIC architecture. Our main objective is to simplify the implementation of an inexpensive one-pass just-in-time (JIT) compiler and to offer a hardware mechanism to reduce the run-time type checking overhead. This work provides a promising intermediate approach to optimize the managed code performance which stands between a pure software solution and a specialized DSP-based architecture.

Although our work targets specifically the EPIC architecture, it may be applied to any other VLIW circuit. Similarly, any stack based language can *de facto* benefit from the proposed hardware assist for stack operations.

In section 2, an overview of some related works aiming an efficient support of MSIL is given. In section 3, we introduce our hardware virtual stack (HVS) based upon the Itanium 2 processors' register stack. In this section, we also describe a conservative implementation of the HVS over EPIC. In section 4, we propose a simple translation mechanism between the MSIL instructions and the EPIC instruction set architecture (ISA) enhanced with our HVS.

Using our performance evaluations, summarized in section 5, we demonstrate that the proposed mechanism is a promising technique to simplify the implementation of inexpensive JIT compilers. We present our conclusions in section 6.

2 Related work

Hardware assist for virtual machine execution is a common approach to either improve pure performance or to provide a flexible execution environment for embedded devices. Java and .NET are both targeted by multiple industrial implementations and academic projects, with a certain advantage to Java essentially due to its earlier availability and adoption. Examples of such specialized processors are the Sun Picojava, Imsys Cjips, aJile or JOP processors/FPGAs [1][2][3][4]. These are good examples of custom designs implementing a dedicated stack engine. For example, The DTC Lightfoot Java Processor Core, has an on-chip data stack composed of 8 32-bit registers (with its tos directly connected to the ALU and a dedicated spill/fill circuit) [5]. The IBM zSeries Application Assist Processors (zAAPs) also provides a dedicated hardware assist to asynchronously execute eligible Java code within the WebSphere JVM under the central processors' control [6]. Stacks, as a hardware abstraction are not only used by Java or .NET. Other languages, such as Forth or C can benefit from such hardware implementation [7][8][9].

Specifically on the embedded end of the spectrum, a strong interest is shown for the Microsoft .NET framework, essentially to offer the framework's design and use flexibility to R&D engineers, yet assuring a fair performance, claimed to be better than software interpretation. JVM and .NET CLR were successfully implemented using FPGAs [10][11][12]. Among these implementations, one is interestingly focusing on the light weighted .NET MicroFramework, a bootable runtime targeting devices with very limited resources [13]. Its MicroBlase soft-core does MSIL interpretation rather than JIT. This is a general trend, which is sometimes enhanced by AOT cross-platform development capabilities.

The solution presented in this paper differs from previous ones in that it corresponds to an in-between approach, leveraging a general purpose enterprise class processor

while providing hardware assist for stack operations in a very conservative way.

3 Hardware virtual registers

3.1 MSIL memory mapping to EPIC

The MSIL execution engine uses three types of local memories and one external memory. Each memory category represents typed data slots [14].

- Argument Table (AT)
- Evaluation Stack (ES)
- Fields – accessed by method (CF – Community of Fields)
- Local Variables Table (LVT)

The AT is directly mapped to the Register Stack Engine (RSE) calling mechanism [15]. The parameters of the *alloc* instruction are computed using the methods' arity *via* the *.maxstack* directive.

The LVT can be either mapped as the AT to the RSE calling mechanism or be mapped into the 32 lower general-purpose (GP) and floating-point (FP) registers. The choice is in the hands of the JIT compilers' implementers. In this study, we will focus on the hardware implementation of the evaluation stack.

3.2 Typed stack

One of the main feature of MSIL is the use of typed stack [14][16]. In other words, the type of the stack entries dynamically changes during the evaluation. There are 7 defined types in MSIL. Table 1 presents our mapping for these types into the native GP and FP registers.

MSIL types	Register mapping
int32	GP (64-bit)
native int	GP (64-bit)
int64	GP (64-bit)
float32 (*80-bit floating-point*)	FP (82-bit)
float64 *(80-bit floating-point*)	FP (82-bit)
& (managed pointer)	GP (64-bit)
ObjectRef (instance pointer to an object)	GP (64-bit)

Table 1: MSIL types and register mapping

3.2.1 Stack dynamic types

To implement the dynamic typing of the ES, we first introduce a hardware virtual stack (HVS). This virtual stack is the representation of the MSIL ES. Nevertheless, as we will disclose shortly, the HVS doesn't store any data. Implementation wise, the virtual stack arbitrarily abstracts the higher 64 GP and FP registers of the Itanium processors – out of 128 GP and FP registers. From this point, the HVS is the only stack visible to the instructions. The HVS also embeds a translation logic which converts the address of an HVS element (also called virtual register) into the address of a physical register in the processor's register files.

By convenience, and for the remainder of the paper, we will call ES the abstracted subset of the GP and FP registers. As we will detail later on, the HVS ensures the correct type of associations between the virtual registers and the physical registers during the stack operations (implicit *push* and *pop*).

The HVS is implemented as a 64-entry table and one 6-bit register called virtual top of stack (*vtos*). Each table entry – also called slot – is 8-bit wide. The slot type (*t* field) is encoded in the 2 high-order bits as shown in Table 2. The third type (t1t0=10) is reserved to handle an in-memory stack. The fourth type (t1t0=11) is reserved and generates an exception if detected.

t1	t0	Type
0	0	General purpose
0	1	Floating-point
1	0	In-memory
1	1	Exception

Table 2: HVS slot type encoding

The residual 6 bits of a slot (the *a* field) encodes the physical address of the associated register. The physical address is provided as an offset into the register file (Figure 1). In complement to the HVS itself, two 6-bit registers stores the address of the top of stack for the GP and FP stacks (called respectively *gtos* and *ftos*).

Finally, two extra registers are used, *rt* (2 bits) and *ra* (7 bits). The *rt* register selects which top of stack (*gtos* or *ftos*) is used during the *push* and *pop* operations (Table 3) while *ra* holds a physical register address (between 0 and 63 + 0x40h).

t field	Top of stack to use
0	gtos
1	ftos
2	gtos
3	Reserved

Table 3: HVS xtos select (x = g | f)

3.2.2 Register allocation and stack operations

A *push* into the HVS begins by checking for a stack overflow exception on *vtos*. If no exception is raised, then *rt* is set with the appropriate type for the pushed data. This type is deduced from the instruction's major opcode (bits 40:37) and the bundle's template bits (bits 0:5) [17]. The value stored into *hvs[vtos]* is computed as ((*rt* << 6) | *xtos*). With *xtos* being either *gots* or *ftos* (systematically noted x = f | g in this article). *ra* (computed as 0x40h + *xtos*) is fed into subsequent pipeline stages. Listing 1 describes the process in pseudo-code.

Note that the *vtos* update happens very early in the address translation process and each address translator is working with a local copy of *vtos*. This mechanism will be detail in section 2.2.4.

```
push() {
  if(!overflow) {
    rt = decod_instruction_type(major_opcode,
```

```
bundle_bits);
      switch(rt) {
      case 00: /* mapped to GP registers */
         hvs[vtos] = ((rt << 6) | gtos);
         ra = 0x40h + gtos++;
         break;
      case 01: /* mapped to FP registers */
         hvs[vtos] = ((rt << 6) | ftos);
         ra = 0x40h + ftos++;
         break;
      case 10: /* mapped to memory */
         /* optional */
         break;
      default:
         ; /* exception */
      }
      vtos++;
   }
}
```

Listing 1: HVS *push* pseudo-code

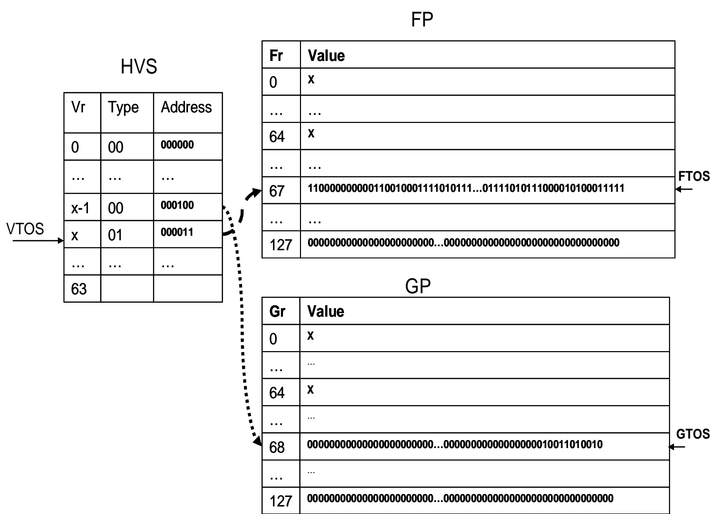

Figure 1: Logical view and usage of the HVS.

A *pop* from HVS starts by a check for a stack underflow exception on *vtos*. If no exception is raised, then *rt* is set with the appropriate type (computed as for the *push* operation by using the instructions' encoding). *hvs[vtos]* is then decoded into register *ra* (receiving the physical register address) and into the type (*t* field). *t* is compared to *rt* for type mismatch (which raises an exception). Relying on the hardware to perform the type checking reduces the associated run-time overhead. Finally *vtos* and *xtos* are updated and *ra* is fed into subsequent pipeline stages. Listing 2 shows the process in pseudo-code. The *spill* / *fill* mechanism of the RSE is used to handle stack over and under flows using the backend memory [18].

```
pop() {
   if(!underflow) {
      rt = decod_instruction_type(major_opcode,
bundle_bits);
#ifdef TYPE_CHECK
      if(rt != (((hvs[vtos]) & (0xC0h)) >> 6)) {
         raise_exception();
      }
#endif /* TYPE_CHECK */
      ra = 0x40h + (hvs[vtos] & 0x3Fh);
      switch(rt) {
      case 00: /* mapped to GP registers */
         --gtos;
         break;
      case 01: /* mapped to FP registers */
         --ftos;
         break;
      case 10: /* mapped to memory */
         /* Optional */
         break;
      default:
         ; /* exception */
      }
      vtos--;
   }
}
```

Listing 2: HVS *pop* pseudo-code

3.2.3 Non-sequential HVS stack operations

As described earlier, the HVS stores in its virtual register's address field the index held by *xtos* register during the *push* operation.

At this stage, we introduce an enhancement to this model to allow a non-sequential allocation and de-allocation of the ES's physical registers. To permit this enhanced mode, two 64-bit vectors are defined, called general-purpose register use vector (GPRUV) and floating-point register use vector (FPRUV). Each bit of these vectors is associated to a physical register of the ES. The vectors are used as bitmaps: a set bit indicates that the associated physical register is used; and a reset bit indicates that the associated physical register is unused and is therefore free to be allocated. These two vectors permit the allocation of a new physical register during the HVS *push* operation in a non-sequential way. Rather than performing *xtos++*, the HVS implements two register address allocation functions (*hvs_xpruv_alloc* with x = g | f). Listing 3 shows the process in pseudo-code.

```
hvs_gpruv_alloc() {
 for(i = 0; i < 64; i++) {
    if(!gpruv[i]) break;
 }
 gpruv[i] = 1;
 gtos = i;
}
hvs_fpruv_alloc() {
 for(i = 0; i < 64; i++) {
    if(!fpruv[i]) break;
 }
 fpruv[i] = 1;
 ftos = i;
}
.
```

Listing 3: HVS *hvs_gpruv_alloc* and *hvs_fpruv_alloc*

The enhanced *push* and *pop* pseudo code using the non-sequential allocation enhancement is given in Listing 4.

```
push() {
   if(!overflow) {
      rt = decod_instruction_type(opcode,
bundle);
      switch(rt) {
```

```
        case 00: /* mapped to GP registers */
           hvs_gpruv_alloc();
           hvs[vtos] = ((rt << 6) | gtos);
           ra = 0x40h + gtos;
           break;
        case 01: /* mapped to FP registers */
           HVS_fpruv_alloc();
           hvs[vtos] = ((rt << 6) | ftos);
           ra = 0x40h + ftos;
           break;
        case 10: /* mapped to memory */
           break;
        default:
           ; /* bypass / exception */
        }
        vtos++;
    }
}
pop() {
   if(!underflow) {
      rt = decod_instruction_type(opcode,
bundle);
#ifdef TYPE_CHECK
        tf = (((hvs[vtos]) & (0xC0h)) >> 6);
        if(rt != tf) {
           raise_exception();
        }
#endif /* TYPE_CHECK */
        ra = 0x40h + (hvs[vtos] & 0x3Fh);
        switch(t) {
        case 00: /* mapped to GP registers */
           gpruv[ra] = 0;
           break;
        case 01: /* mapped to FP registers */
           fpruv[ra] = 0;
           break;
        case 10: /* mapped to memory */
           break;
        default:
           ; /* bypass / exception */
        }
        vtos--;
    }
}
```

Listing 4: Non-sequential HVS *push* and *pop*

3.2.4 HVS use by EPIC instructions

Most EPIC instructions are triadic. Only three floating-point instructions (*fma*, *xma* and *fselect*) have a four register form. Let's review the HVS use by the *fma* instruction as a case study. The *fma* instruction's form is:

(qp) *fma.pc.sf f1 = f3, f4, f2.*

The operation carried-out is f1 = (f3 x f4) + f2. All the register operands are encoded using 7 bits.

When solicited by an *fma* instruction, the HVS translates f1, f2, f3 and f4 (let's call them fx) initial 7-bit values called here virtual register indexes, into fx' 7-bit values, called here physical register indexes. As explained, physical register indexes point to registers in the FP register file. The translation takes place only if the msb of fx is set. If the msb of fx is not set, then no translation is performed. It is noticeable that only the msb value has to be set by the compiler when encoding for the instruction.

We will signal it using the 1xxxxxx value in the remainder of this paper. It is also recommended – for clarity purpose – that the compiler uses a unique and documented value when targeting the ES (the HVS translation).

Let's assume that all the registers of an *fma* instruction are virtual and needs to be translated. In this case f2, f3 and f4 are all translated as described in the *pop* operation (enhanced or not), and f1 is translated as described in the *push* operation. The translated register addresses are passed down the next pipeline stage. The process is summarized in Listing 5. Note that the compiler can decide not to store the result of the *fma* instruction in the ES, but rather decide to transfer it into any register not used by the ES. To do so, it suffices to clear f1's msb.

```
if(msb(f4 == 1)) {
   f4' = pop();
}
if(msb(f3 == 1)) {
   f3' = pop();
}
if(msb(f2 == 1)) {
   f2' = pop();
}
if(msb(f1 == 1)) {
   f1' = push();
}
```

Listing 5: HVS *fma* instruction registers

From an implementation point of view, there is one translator for each stack (general-purpose, floating-point and the optionally in-memory), and each of them is able to carry-out up to four register address translations in parallel. Each address translation unit works with a local *vtos*. Each local *vtos* value is quickly computed at the instruction decoding stage. This latency is absorbed by consecutive stages. Listing 6 shows the process applied to the *fma* instruction.

```
/* fast serial vtos update / copy */
if(msb(f4 == 1)) {
   vtos4 = vtos--; call
register_address_translator4();
}
if(msb(f3 == 1)) {
   vtos3 = vtos--; call
register_address_translator3();
}
if(msb(f2 == 1)) {
   vtos2 = vtos--; call
register_address_translator2();
}
if(msb(f1 == 1)) {
   vtos1 = vtos++; call
register_address_translator1();
}

/* parallel address translation */
register_address_translator4() {
   f4' = pop(); /* using vtos4 */
}
register_address_translator3() {
   f3' = pop(); /* using vtos3 */
}
register_address_translator2() {
   f2' = pop(); /* using vtos2 */
}
```

```
register_address_translator1() {
   f1’ = push(); /* using vtos1 */
}
```

Listing 6: HVS four register address translation process with early *vtos* update and copy

The translation mechanism from virtual register address into physical register address described for the *fma* instruction is generalized to any EPIC instruction using at least one register as source or destination operand. From an implementation point of view, a multi port memory is used to allow simultaneous stack accesses.

3.2.5 HVS use by instructions bundles

In this section we review the HVS use during bundles' execution. EPIC architecture defines a bundle as a group of three instructions encoded in 128 bits (Table 4).

Bits	127-87	86-46	45-5	4-0
Content	instruction slot 2	instruction slot 1	instruction slot 0	template
Size (in bits)	41	41	41	5

Table 4: Encoding of an instruction bundle

An EPIC processor can execute several bundles in parallel. For example, the Itanium 2 processor can schedule for execution up to two bundles in parallel during the same clock cycle. If there are no dependencies between the six instructions of two consecutive bundles, then all the instructions are executed simultaneously, assuming that the data and the execution units are available.

Let's first review the HVS use by a single bundle. If none of the three instructions of a bundle is using the HVS, then there is no impact on the bundle's execution at all. Because of the sequential nature of stack operations (*push* / *pop*), any instruction that pushes a result into the HVS will induce an implicit stop after its instruction slot [15]. It is recommended that the compiler enforces the use of explicit stop bits when building the bundles holding instructions using the HVS to ensure the correctness of the algorithms.

Yet, we are introducing a second execution mode which doesn't enforce implicit stops (a.k.a. non-blocking mode). In this mode, the target stack-level of a *push* into the HVS is determined by the instructions' rank – or slot – in the bundle, which is computed using the bundle's template bits.

Listing 7 shows two sample bundles using the explicit bundle marking. The first bundle (lines 1 to 5) is composed by an integer ((qp) *add* r1 = r2, r3) and a floating-point instruction (*fma*). The second bundle (lines 6 to 10) is composed by two *add* instructions.

```
 1: { .mfi /* template bits */
 2:    add r1xxxxxx = r1xxxxxx, r1xxxxxx /*slot 0*/
 3:    fma f1xxxxxx = f1xxxxxx, f1xxxxxx, f1xxxxxx
/* slot 1 */
 4:    nop.i                            /*slot 2*/
 5: }
 6: { .mii /* template bits */
 7:    ld4 r1xxxxxx = [r1xxxxxx]
/* slot 0 */
 8:    add r1xxxxxx = r1xxxxxx, r1xxxxxx
/* slot 1 */
 9:    add r1xxxxxx = r1xxxxxx, r1xxxxxx
/* slot 2 */
10: }
```

Listing 7: Two sample bundles

By convention, the HVS will perform the register address translation starting with the instruction in slot0 and continuing up to the end of the parallel execution window (end of second bundle or first explicit / implicit stop). The result of our two sample bundles' execution is summarized in Table 5, Table 6 and Table 7. To improve applications' ILP and efficiently use of the underlying architecture, it is the compiler's responsibility to leverage the non-blocking mode. It is also the compiler's responsibility to use or not the HVS. For the former, a first bit is used to signal the HVS capability (to be checked at runtime *via* the CPUID instruction). A second bit is used in an MSR to dynamically activate or deactivate the HVS.

By setting / resetting this bit at run time, the processor can be reconfigured into its default mode, bypassing the HVS logic. When the code requires the use of the HVS, the VM can, *via* the MSR, re-activate the HVS.

instructions	pop count	push count
add	2	1
fma	3	1
nop	0	0
ld	1	1
add	2	1
add	2	1
total	***10***	***5***

Table 5: Two bundles scheduled for parallel execution

Stacks initial state		
HVS	**GP**	**FP**
*		
a		
b		
c	*	
d	a	
e	b	
f	f	
g	g	*
h	h	c
i	i	d
j	*j*	*e*

Table 6: Initial HVS state. * represents xtos. The sequential stack allocation notation is used for clarity.

Stacks final state		
HVS	**GP**	**FP**
*		
a + b	*	
c.d + e	a + b	
[f]	[f]	
g + h	g + h	*
i + j	*i + j*	*c.d + e*

Table 7: Final HVS state. * represents xtos. The sequential stack allocation notation is used for clarity.

4 Translating MSIL into EPIC code

The .NET IAS can be mapped, almost 1:1, into the EPIC IAS [17][16][19]. Our 1:1 single-pass software translator from MSIL into EPIC assembler uses this property. Let's study the simple function that computes the surface of an irregular triangle using the Heron formula in double precision. Its disassembly is partially given in Listing 8.

```
01: IL_0000:  ldarg.0          //stack=a
02: IL_0001:  ldarg.1          //stack=b
03: IL_0002:  add              //stack=a+b
04: IL_0003:  ldarg.2          //stack=c
05: IL_0004:  add              //stack=a+b+c
06: IL_0005:  ldc.r8 0.5       //stack=0.5
07: IL_000e:  mul              //stack= 0.5*(a+b+c)
08: IL_000f:  stloc.0     // x = 0.5 * (a + b + c)
09: IL_0010:  ldloc.0  etc…
10: IL_0011:  ldarg.0
11: IL_0012:  sub
12: IL_0013:  ldloc.0
13: IL_0014:  mul
14: IL_0015:  ldloc.0
15: IL_0016:  ldarg.1
16: IL_0017:  sub
17: IL_0018:  mul
18: IL_0019:  ldloc.0
19: IL_001a:  ldarg.2
20: IL_001b:  sub         // S ^ 2 = x * (x - a) *
21: IL_001c:  mul         //     (x - b) * (s - c)
```

Listing 8: Partial disassembly for the Heron function

The result of this inexpensive one-pass translation using the HVS is given in Listing 9. With the exception of the loading of the 0.5 constant which requires two instructions with the Itanium processors (lines 5 and 6), each MSIL instruction translates into one EPIC instruction and the use of the HVS makes the translation straight forward.

```
01: fadd.d    f64=f8,f0 ;;          // ldarg.0
02: fadd.d    f64=f9,f0 ;;          // ldarg.1
03: fadd.d    f64=f64,f64 ;;        // add
04: fadd.d    f64=f10,f0 ;;         // ldarg.2
05: fadd.d    f64=f64,f64           // add
06: movl      r2=0x3fe0000000000000 ;;
07: setf.d    f64=r2 ;;             // ldc.r8 0.5
08: fma.d     f64=f64,f1,f64 ;;     // mul
09: fadd.d    f32=f64,f0 ;;         // stloc.0
10: fadd.d    f64=f12,f0 ;;         // ldloc.0
11: fadd.d    f64=f8,f0 ;;          // ldarg.0
12: fsub.d    f64=f64,f64 ;;        // sub
13: fadd.d    f64=f32,f0 ;;         // ldloc.0
14: fmad.d    f64=f64,f1,f64 ;;     // mul
15: fadd.d    f64=f32,f0 ;;         // loadloc.0
16: fadd.d    f64=f9,f0 ;;          // loadarg.1
17: fsub.d    f64=f64,f64 ;;        // sub
18: fma.d     f64=f64,f1,f64 ;;     // mul
19: fadd.d    f64=f32,f0 ;;         // ldloc.0
20: fadd.d    f64=f10,f0 ;;         // ldarg.2
21: fsub.d    f64=f64,f64 ;;        // sub
22: fma.d     f64=f64,f1,f64 ;;     // mul
```

Listing 9: 1:1 translation of MSIL into EPIC

The translation uses the following conventions. The input arguments *arg.0*, *arg.1*, *etc.* translates into [*f8-f15*] and [*r32-r63*] registers, based on their type. *loc.0*, *loc.1*, *etc.* are translated, depending on their type, translated into the [*f32-f63*] and [*r14-r27*] scratch registers.

Note that in this example *fadd* is used as a move instruction. *fadd* and *fsub* are respectively synonyms of *fma* and *fms*. This makes sense since the Itanium 2 processor has 11 issue ports, among those are two highly optimized floating-point ones. For clarity, we can signal that *f0* = 0.0 and *f1* = 1.0.

At no additional expense the translator performs, during its single translation pass, several optimizations to enhance the use of the processor's resources and the HVS. Among those, it fuses sequences of stack operations, it refers arguments and local variable registers directly and when applicable, schedules store and load operations to the same clock cycle (Listing 10). A look-ahead window is used to detect independent instructions.

At the end of the code generation, the instructions and the scheduled clock cycles counts are reported for comparison. These code quality metrics are *lower is better*.

```
01: fadd.d    f64=f8,f9 ;;  // ldarg.0,ldarg.1,add
02: fadd.d    f64=f10,f64         // ldarg.2,add
03: movl      r2=0x3fe0000000000000 ;;
04: setf.d    f64=r2 ;;           // ldc.r8 0.5
05: fma.d     f64=f64,f1,f64 ;;   // mul
06: fadd.d    f32=f64,f0 ;;       // stloc.0
07: fadd.d    f64=f12,f0 ;;       // ldloc.0
08: fsub.d    f64=f64,f8 ;;       // ldarg.0,sub
09: fma.d     f64=f64,f1,f32 ;;   // ldloc.0,mul
10: fsub.d    f64=f32,f9 ;;  //ldloc.0,ldarg.1,sub
11: fma.d     f64=f64,f1,f64 ;;   // mul
12: fsub.d    f64=f32,f10 ;; //ldloc.0,ldarg.2,sub
13: fma.d     f64=f64,f1,f64 ;;   // mul
```

Listing 10: Fused translation of MSIL into EPIC

In addition to comparing our translator versions (1:1 and fused), we are benchmarking ourselves with state-of-the-art compilers (the testing methodology is detailed in section 5.1 – Listing 11).

```
01: fma.d     f15=f8,f1,f9
02: movl      r2=0x3fe0000000000000 ;;
03: setf.d    f14=r2 ;;
04: fma.d     f13=f15,f1,f10 ;;
05: fma.d     f12=f14,f13,f0 ;;
06: fms.d     f11=f12,f1,f8
07: fms.d     f9=f12,f1,f9 ;;
08: fms.d     f8=f12,f1,f10 ;;
09: fma.d     f7=f12,f11,f0 ;;
10: fma.d     f6=f7,f9,f0 ;;
11: fma.d     f8=f6,f8,f0 ;;
```

Listing 11: Code generated by an optimizing compiler

5 Experimental results

5.1 Test methodology

This section presents the methodology used to evaluate the potential performance impact of the HVS on a simplistic JIT translator and the benchmark used to compare our initial approach versus state-of-the-art JIT and native compilers. We tested the Kasumi algorithm [20].

Kasumi is a block cipher algorithm that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. This application by its profile provides a good first approximation for integer dominated branchy code. It is therefore representative of the enterprise-class applications targeting systems based on the Itanium family processors. A larger number of applications will be evaluated in the future.

We have implemented the confidentiality algorithm f8 [21] –built upon Kasumi– in managed C++ and in a reference native C++ code. Both codes are strictly identical in their computational kernels. The f8 implementation has an integer-branchy profile. The relative importance of the key functions is shown in Figure 2.

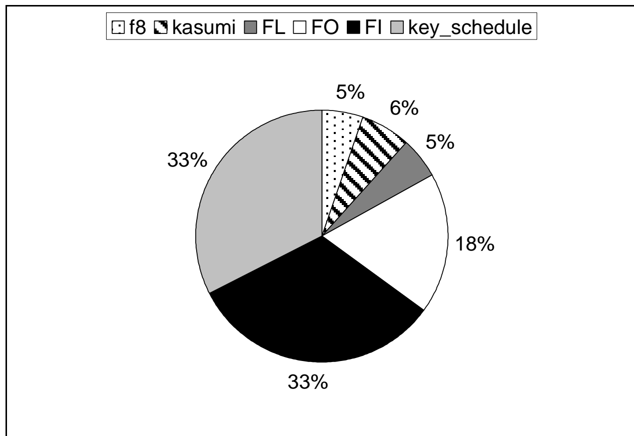

Figure 2: Retired instructions distribution

To collect the data for our comparisons, we first generated the MSIL code for the application by compiling it for the CLR [22][23].

The MSIL code is then translated into EPIC assembler using our 1:1 and the fused translator targeting the HVS. To collect the JITed data, we are using the VTune performance analyzer [24][25], dumping the disassembly of the JITed and compiled methods' run traces.

Finally, to compare our translations and JITed code to the best code generation techniques, the reference C++ code was compiled using the highest optimization level (O3, IPO and PGO enabled) [26].

As per our benchmark metrics, we collect the instruction count – excluding *nops* – and the scheduled clock cycles. Both benchmark metrics are *lower is better*. Figure 3 and Figure 4 compares the quality of the code generated by the translators and the reference JIT compiler, where Figure 5 depicts the average IPC for all code generation methods. Instructions count doesn't refer to an execution trace but to the code generated by the compilers and the translators.

Although the code generation time is an essential factor, it was impossible to accurately record the JIT compilation duration with our experimental setup. However, the translator's run time is at least a magnitude order less than the JIT and the native compiler's run times.

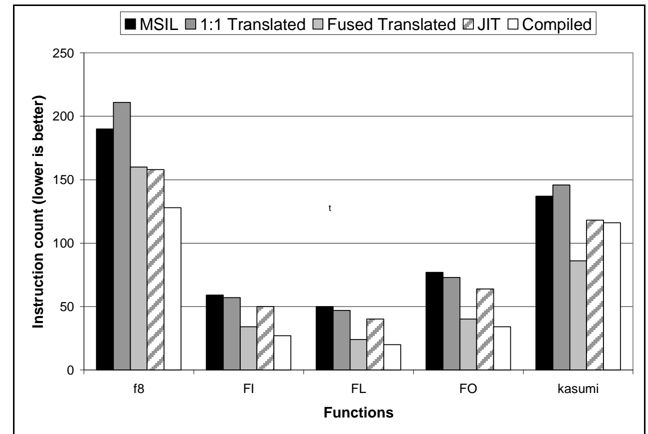

Figure 3: Generated instruction count

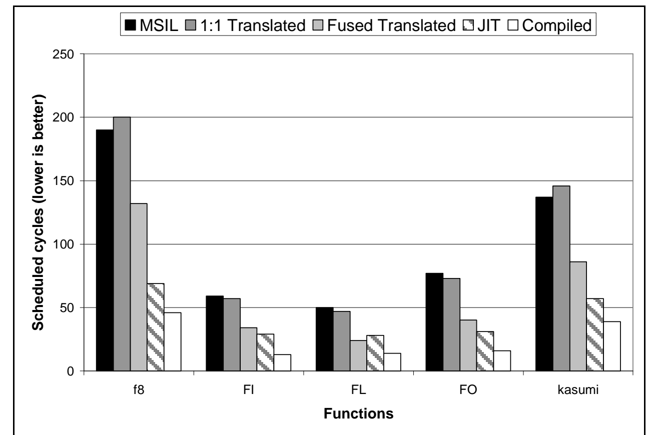

Figure 4: Scheduled clock cycles

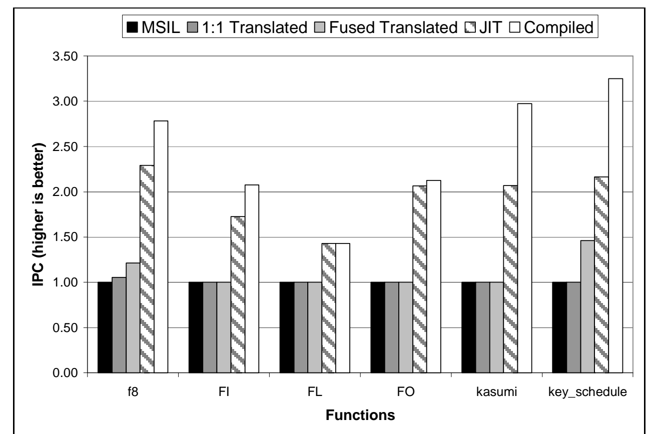

Figure 5: Average IPC (ideally the IPC is equal to 6)

5.2 Analysis

Overall, the number of instructions generated by the 1:1 translator is, as expected, equivalent to the number of IL instructions. If in most of the cases, the required instruction

count is lower or equal to the MSIL reference (-0.17% in average), several constructs used for the translation requires from 1:2 up to 1:17 instructions generation ratio to satisfy the EPIC architecture. Branching and compares using predicate registers, loading floating-point constants or performing divisions are such cases. For example, in *key_schedule*, the translators are unable to perform strength reduction and translate divisions by even values into shift operations and have to perform divisions (which is not implemented in hardware).

The fused translator reduces the number of instructions generated by 65.9% in average [-18.5%;-108.3%]. On the major functions in term of instructions retired during run-time (84% of the total retired instructions), the fused translator outperforms the JIT compiler, and is close to be par with the native compilers (Figure 3). Our tests show that multiple improvements in the generated code's quality are possible at the expense of additional passes. For example, vector processing using the SIMD capable execution units of the processor can reduce the instructions count by 8 in the computational kernel of *key_schedule* and bring us in par with the native compiler.

In complement of the instruction count we also consider the scheduled clock cycles to evaluate the generated code's quality. This count is the code generator's best scheduling and is not necessarily equal to the number of cycles required to execute the instruction bundles as they may introduce extra clock cycles on data dependencies when accessing the memory sub-system (caches included). The 1:1 translator doesn't take any advantage form the VLIW architecture of EPIC and schedules 1 instruction per clock cycles. As for the instruction counts, some variations are noticed due to generation of additional code, which doesn't depend on the evaluation stack and hence, can be scheduled more aggressively. The fused translator performs basic instruction scheduling optimizations while it fuses stack operations. A look-ahead window is used to explore the IL code sequence searching for independent load and store instructions. Those can be advantageously scheduled in the same pair of bundles for the same clock cycle [23]. This is mainly noticeable in *key_schedule* where sequences of two loads and two stores are scheduled per clock cycle. On average, the number of scheduled cycles is reduced by 81% [-43.9%;-108.9%]. Because of our short look-ahead window (12 IL instructions), the fused translator doesn't allow for deep ILP improvements (Figure 4). This accounts for the strong IPC deficit in the translated code when we compare it to the code generated by the optimizing [1.43-2.97X] and the JIT [1.43-2.07X] compilers (Figure 5).

Although the non-blocking mode of the HVS could allow for additional improvements of the ILP by identifying and scheduling independent processing paths together, we did not perform such optimizations in the scope of this work as it would require a multi-pass approach, moving us away from our initial objective

6 Conclusions

We defined an MSIL hardware evaluation stack (Hardware Virtual Stack, or HVS) based on the EPIC architecture. As it is defined, the HVS also provides a hardware support to offload the run-time type checking performed by the CLR. In addition, several enhanced modes are proposed for the HVS to allow for non-sequential register allocation and a non-blocking parallel stack access. The – optional – use of the HVS allows the creation of very simple, one-pass MSIL code translator into EPIC assembler.

Based on our testing and benchmarking versus state-of-the-art optimizer and JIT compilers, our translator, applying a limited number of optimization techniques, performs fairly. Most of the future improvements need to focus on the IPC increase in the generated code. The non-blocking mode of the HVS is the path that we plan to explore to achieve this goal in the future.

Although we are not capturing the advantage of a short run-time of our translators and the performance benefit provided by offloading the run-time type checking, our data allows us to still consider that the HVS is a promising technique and meets its objectives.

7 References

[1] 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.

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

[3] D. S. Hardin, "*aJile Systems: Low-Power Direct-Execution Java Microprocessors for real-Time and Networked Embedded Applications*", www.ajile.com.

[4] M. Schoeberl, Institute of Computer engineering Vienna University of Technology, Austria, "*JOP – a Real-time Processor*", 2006

[5] Digital Communication Technologies, "*Lightfoot 32-bit Java Processor Core*", www.dtcl.com.

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

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

[8] J. Tayeb, S. Niar, *"Adapting EPIC Architecture's Register Stack for Virtual Stack Machines"*, 9th EUROMICRO Conference on Digital System Design (DSD'06), pp. 204-210, 2006.

[9] J. Tayeb, S. Niar, *"Optimizing Intel EPIC/Itanium2 Architecture for Forth"*, European Forth Conference, 2006.

[10] Srinath S, Srinivasan.T, VidhyaBhushan.M, Ranjani Parthasarathi, Department of Computer Science, Anna University, "*Implementation of .NET CLR on FPGAs*", 2005.

[11] L.V. Nagendra Kumar, International Institute of Information Technology, Gachibowli, Hyderabad, India, "*JVM Implementation in FPGAs*", B.Tech final year Project report, 2002.

[12] D. Ragozin, A.Umnov, M.Shuralev, M.Sokolov, A.Eltsov, *"The CIL Hardware Processor: An Implementation of CIL Code Execution Engine"*, Technical report #5 for RFP2 Hardware CIL Processor project, Nizhny Novgorod State University, 2006.
[13] M. Sneha, D Sri Charanya, R. Usha Bhuvaneswari, "*Hardware Support for .NET MicroFramework CLR Components*", Department of Computer Science, College of Engineering, Guindy, Anna University, hpic2006
[14] J. Gough, *"Compiling for the .NET Common Language Runtime (CLR)"*, Prentice Hall, 2002.
[15] Intel Corporation, *"Intel® Itanium® Architecture Software Developer's Manual Volume 2: System Architecture"*, Revision 2.2, 2006.
[16] S. Lidin, *"Inside Microsoft .NET IL Assembler"*, Microsoft Press, 2002.
[17] Intel Corporation, *"Intel® Itanium® Architecture Software Developer's Manual Volume 3: Instruction Set Reference"*, Revision 2.2, 2006.
[18] Intel Corporation, *"Intel® Itanium® Architecture Software Developer's Manual Volume 1: Application Architecture"*, Revision 2.2, 2006.
[19] Intel Corporation, *"Itanium® Software Convention and Runtime Architecture Guide"*, 2001.
[20] 3rd Generation Partnership Project – Technical Specification Group Services and System Aspects (3GPP TS 35.202 V3.1.1), *"Technical Specification 3G Security: Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification"*, 2001
[21] 3rd Generation Partnership Project – Technical Specification Group Services and System Aspects (3GPP TS 35.204 V3.1.2), "*Technical Specification 3G Security: Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 4"*, 1999.
[22] Microsoft® .NET Framework 3.5.
[23] Microsoft® C/C++ Optimizing Compiler Version 15.00.20706.01 for Itanium.
[24] S. Niar, J. Tayeb, *"Les processeurs Itanium: Programmation et optimization"*, Eyrolles, 2005.
[25] Intel® VTune™ Performance Analyzer 9.0. Build:24052.
[26] Intel® C++ Compiler for Itanium®-based applications. Version 9.1 Build 20061105. Package ID: W_CC_C_9.1.033.