SCISM vs IA-64 tagging: differences/code density effects

Georgi Gaydadjiev and Stamatis Vassiliadis

Computer Engineering Lab, EEMCS, TU Delft, The Netherlands,
E-mail:{G.N.Gaydadjiev,S.Vassiliadis}@EWI.TUDelft.NL
http://ce.et.tudelft.nl/

Abstract. In this paper we first present two tagging mechanisms; the SCISM and IA-64; thereafter we describe the mapping of IA-64 ISA to a SCISM configuration without changing or reassigning the IA-64 instructions to preserve the original architectural properties. Under this limiting SCISM scenario, opcode reassignment will improve even more the SCISM performance, it is shown that SCISM tagging will significantly improve (between 21 and 29%) static code density. The results are based on analysis of various SPECINT2000 executables.

Keywords: Instruction Tagging, Instruction Level Parallelism, SCISM, IA-64.

1 Introduction

Tagging has been used extensively by microarchitects and designers as an efficient mechanism to facilitate implementation and potentially improve the performance of processors. Tagging for example has been used to enumerate and manage the hardware resources [1], to handle interrupts, e.g. [2], speculative execution (see for example [3], and to facilitate concurrent instructions routing see for example [4]. Instruction tagging for instruction level parallelism has been introduced for two main reasons namely: to reduce the complexity (and the cycle time), mostly the decode stage, of a pipelined machine implementation, and to potentially improve instruction level parallelism (ILP).

In this paper we analyze and compare the two ILP tagging mechanisms, SCISM [5] (the first known machine organization that employs tagging with the mentioned ILP characteristics) and IA-64 [6], [7] tagging mechanisms and provide evidence suggesting that the SCISM tagging provides some benefits when compared to IA-64 tagging. In particular we investigate and show the following: We provide evidence indicating that the SCISM tagging is a superset of the IA-64. We consider the side effects of tagging on code densities and show that the SCISM tagging of IA-64 instructions regarding code densities is clearly superior to the IA-64 for static code. In particular it is shown that the SCISM tagging reduces the IA-64 code size for SPECINT2000 benchmarks between 21% and 29%.

The paper is organized as follows. Section 2 gives a short description of IA-64 architecture [1] with the main focus on the *template* bit field role. In addition, the SCISM organization is described with emphasis on tags and their functionality. Section 3 maps the IA-64 instruction set architecture to SCISM and shows how the original IA-64 aspects are preserved. In the same section the results concerning the static code density are discussed and the discussion is concluded.

[1] We note that in this paper we use the original definition of the term of architecture as described by [8].

2 The IA-64 and SCISM tagging

IA-64 uses *bundles* as its compound instruction format. A bundle consists of three instruction slots and a template field. Each bundle in IA-64 is 128-bits long. Figure 1(a) shows the bundle's format. I_0, I_1 and I_2 represent the three instructions (41-bits each), while $template$ (tag) is a 5-bit wide field. The template information is used for decoding, routing (dispersal) and ILP. Instruction groups can be seen as chained bundles in the absence of stops. The boundaries between instruction groups correspond directly to the *instruction level parallelism* (ILP) in a particular IA-64 implementation.

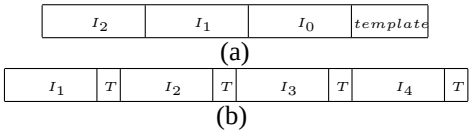

Fig. 1. IA-64 and SCISM bundle formats

IA-64 has five instruction slot types corresponding to the different execution unit types - Memory (M), Integer (I), Floating-point (F), Branch (B) and Long (extended) (L+X), or in shorthand (M, I, F, B, and L). IA-64 instructions are divided among six different instruction types - ALU (A), Memory (M), Integer (I), Floating-point (F), Branch (B) and Long (extended) (L+X), abbreviated as (A, M, I, F, B, and L). An interesting detail is that instruction of A-type, e.g. integer add, can be scheduled to either I or M execution unit. The L+X type uses two instruction slots and executes on I-unit or on B-unit. Due to the limited number of bits not all instruction triples are supported. There are 12 basic template types (each with two versions with stop on the bundle end or not): MII, MI_I, MLX, MMI, M_MI, MFI, MMF, MIB, MBB, BBB, MMB and MFB, where "_" (underscore) indicates a stop inside the bundle (not at the bundle boundaries).

In SCISM, instructions are *categorized* according to hardware utilization not op-code description. An obvious implication of this is that the number of rules needed to determine parallel execution depends on the number of categories, rather than on the number of individual instructions. Given that a category comprises of multiple instructions used by a single hardwired unit, the differences among category members are considered as "trivial" and are resolved by the hardware by means of some control signal or by minor hardware modifications. For example in an implementation [9] a set of fourteen IBM 370 ISA [10] operations is presented that belong to a single category and are executed by the same hardware (ALU) (see Figure 2). Another categorizations are obviously possible. Two different tagging mechanisms have been reported [11], [5], [12]. The first mechanism [5] requires $\lceil log_2(n) \rceil$ bits, with n being the number of instructions to be executed in parallel. The second (original) mechanism requires only one additional bit as depicted in Figure 1(b) for an example 4 in-

OPERATION	ALU Function	Operand Representation	Requires
Load Complement (LCR)	32-b signed addition	Two's complement	Adder
Load Positive (LPR)	32-b signed addition	Two's complement	Adder
Load Negative (LNR)	32-b signed addition	Two's complement	Adder
Load Register (LR)	32-b signed addition	Two's complement	Adder
Load and Test (LTR)	32-b signed addition	Two's complement	Adder
AND (NR)	bitwise logical AND	Binary	Logical
OR (OR)	bitwise logical OR	Binary	Logical
EXCLUSIVE-OR (XR)	bitwise logical EX-OR	Binary	Logical
Add (AR)	32-b signed addition	Two's complement	Adder
Subtract (SR)	32-b signed addition	Two's complement	Adder
Add Logical (ALR)	32-b unsigned addition	Unsigned Binary	Adder
Subtract Logical (SLR)	32-b unsigned addition	Unsigned Binary	Adder
Compare Logical (CLR)	32-b unsigned addition	Unsigned Binary	Adder
Compare (CR)	32-b signed addition	Two's complement	Adder

Fig. 2. RR-Format Loads, Logicals, Arithmetics and Compares operations [9]

structions wide parallel machine, with I_1, I_2, I_3 and I_4 being the original instructions. In Figure 1(b), all instructions I_j are in their original form and $T \epsilon \{0,1\}$ represent the tags. The SCISM approach implies full binary compatibility, allowing straight-forward legacy code execution and parallelization. An interesting implicit property associated with the SCISM tagging is that it is allowed, contrary to the IA-64 tagging, to branch in the middle of a compound instruction allowing code compaction (complete elimination of nops and removing the need of branch alignment) [5]. Only tags are added to code thus if SCISM tagging is applied to existing code the original code remains unchanged. As a consequence there are no side effects such as branch target calculations.

3 Tagging effects on code size

This section begins by showing how IA-64 instructions can be mapped onto SCISM without strict code mapping (no opcode space re-assignment). This straight-forward mapping is not an optimal approach for SCISM due to the shared major IA-64 opcodes, but is a quick way to demonstrate the SCISM potential and create a base for comparison. It should be taken into account that this is also the worst-case scenario with respect to SCISM when investigating binary code density. To transform IA-64 to SCISM code a three-way SCISM organization is assumed. This is to create an IA-64 bundle which corresponds to a SCISM compound instruction with a length of three. Please note that the discussion on code density differences is unrelated to any particular IA-64 implementation. To clarify this: the Itanium2 dispersal window (two bundles) corresponds to a six-way SCISM compound instruction leaving the code size differences between the two approaches unchanged. The SCISM organization by its definition is not restricted to certain number of instruction combinations (24 out of 32 possible when using 5 bits) while IA-64 is. The three-way SCISM compounding requires three tag bits for stop indication (see Section 2), leaving two out of five IA-64 template bits unused. On the other hand, the template removal will require additional information about the functional unit to be added to each individual instruction. In IA-64 all of the instructions are executed by one of the four execution units types: M, I, F or B. The two ”remaining” IA-64 instruction types (A-type and X-type) are also executed by one of those execution units types (A-type by I or M and (X-type) by I or a B unit). This is why the additional bits are coupled to the designated functional units instead of the instruction types. This requires two additional bits for each basic instruction (or 6 for the total compound instruction). The SCISM instruction format for IA64 is depicted in Figure 3 (a). As stated earlier three single bit tags are needed to express the IPL (shown as T in the figure). In addition, another two bits for routing (rr) per instruction are used to provide information about the targeted functional unit. Stated differently this is a 3-bit SCISM tagging [11]. The three 41-bit long IA-64 instructions are unmodified in their original form. Putting it together, SCISM instructions become 44-bit long (including tagging), hence the three way compound instruction will become 132 bits.

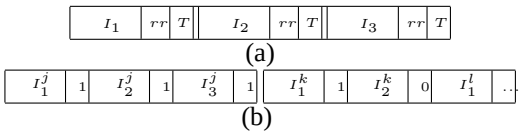

Fig. 3. Instruction format and bundle chaining

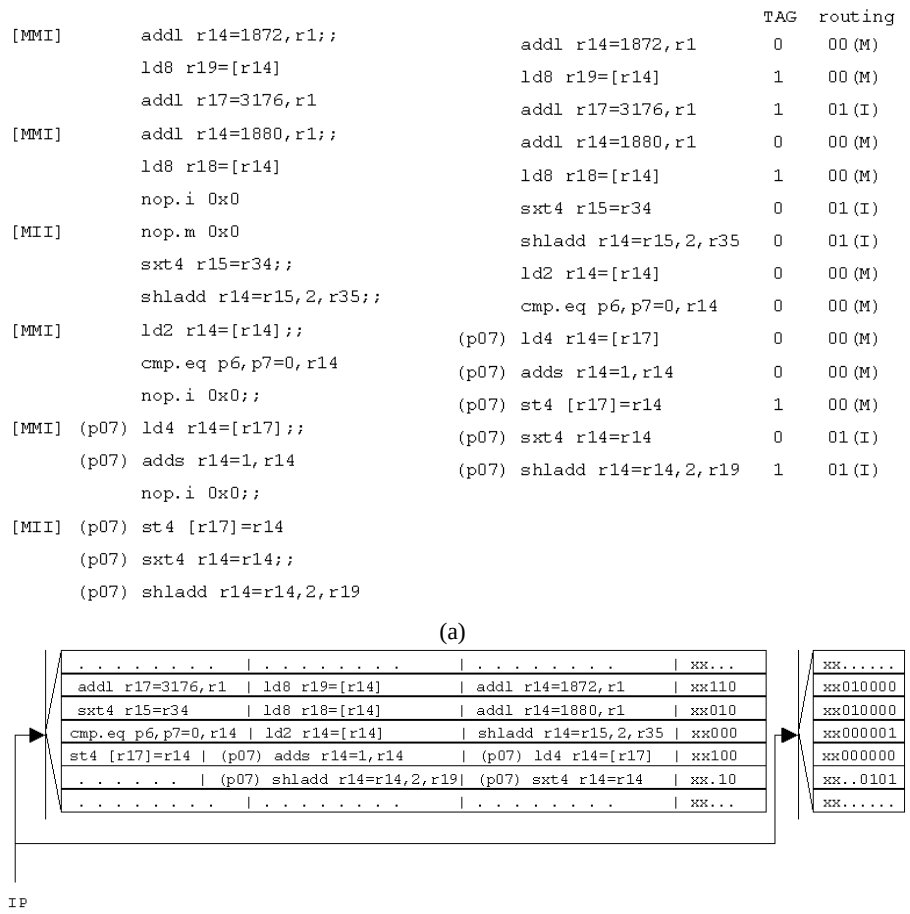

Fig. 4. IA-64 and SCISM code (*build_tree* function of gzip)

The template bits are not needed, since bits are added to indicate the position where a compounding is ending. When ”wider” than the compound instruction implementation is used can be easily implemented in SCISM. Figure 3(b) shows an example of how 5 parallel instructions can be marked in the proposed 3-way SCISM organization, where I^j, I^k and I^l represent three subsequent SCISM compound instructions. To clarify the discussion above Figure 4(a) depicts a piece of the *build_tree* function code from the IA-64 binary of the *gzip* executable. In Figure 4(a) the left column shows the original IA-64 code, next column is the equivalent SCISM code, the TAG column represents the tagbits, and the routing information is shown in the last column. The encoding is for: memory unit (M) = 00, and integer unit (I) = 01. Figure 4(b) shows a potential memory organization for IA-64 compounded instructions where the compound instructions are 136 bits long (the original 128 bit bundle plus an additional byte). Please note that this is one out of many implementations possible that can facilitate the proposed organization. The Template information of the original IA-64 bundle is replaced by the tags corresponding to the three instructions (filled with two don’t care bits - ’x’), while the additional byte is addressed in parallel with the modified IA-64 bundle to access the

routing information (along with the two additional spare bits). Furthermore, as in the case of S/370 example, except for the tagging, IA-64 instructions have not been modified preserving all the instruction properties. In order to find the improvement in code size we investigated the SPECINT2000 executables. Since the compiler optimizations may play significant role in the IA-64 approach, the effects of different compilers on code size where investigated in [13]. The differences on code sizes produced by different compilers, e.g. gcc and Intel where found marginal, so the results in this paper are independent on the compiler technology. The benchmarks where compiled using the

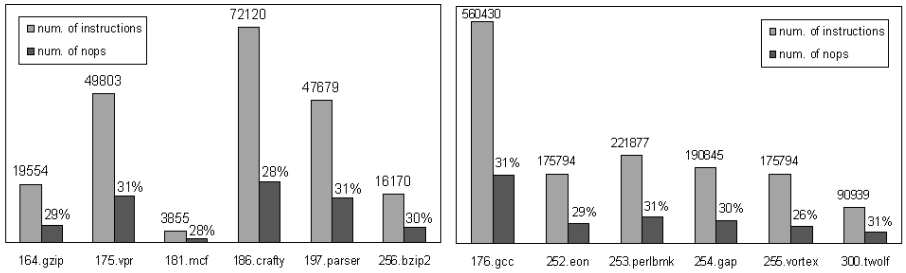

Fig. 5. IA-64 NOP utilization SPECINT2000

CPU2000 default makefiles (optimization levels). The compilation was performed inside *Native User Environment* (NUE) developed at the Hewlett-Packard Labs [14]. This environment emulates a Linux/ia64 system, more precisely Itanium2, and was considered sufficient since the static binary code investigation was the primer concern. We considered only the code segments of the benchmark executables, leaving all other program segments out. The results are presented in Figure 5. The first column shows the total number of instructions involved, e.g. the complete code segment in instructions (not bundles). The second column in each pair shows the percentage of nop instructions found. It was found that for all of the benchmark executables approximately one third of the operations are nop operations. The 255.vortex benchmark was found as the one with the lowest nop count (26%), however this is an exception.

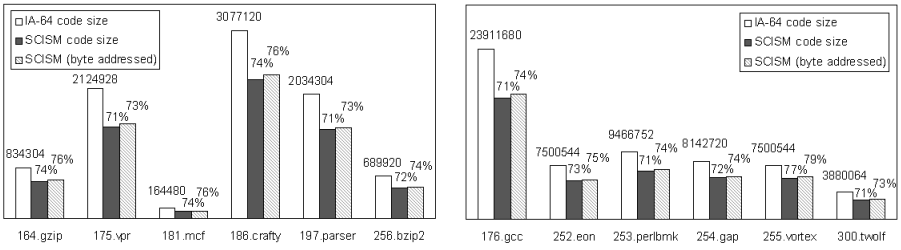

Fig. 6. IA-64 vs SCISM code size SPECINT2000

The comparison on the code segment size between the IA-64 (Itanium2) and the SCISM is presented in Figure 6. The IA-64 results where estimated as follows: the total number of bundles (instead of instructions) was used, since the bundle size is predetermined to 128 bits it is a simple procedure to determine the code segment size in number of bits. In case of SCISM, the number of instructions was used and the worst-case in-

struction length of 44-bits was assumed. An even more restrictive scenario for SCISM assumes instructions to be byte rather than nibble addressed. This can be done with the addition of four more tag bits per bundle. Strictly speaking these four bits are not needed and can be used to improve performance and/or the hardware design. They will add however to the storage requirements. The results are presented in Figure 6, where the first column represents the number of bits (of the code segments) for the original benchmark executables. The second column shows how the code segment size when the SCISM approach is applied and the ratio is expressed in percent of the original size. The third column represents the byte addressed SCISM scenario when the instruction bundles are expanded to 136-bits by adding a second nibble. It can be seen that IA-64 code size for the SPECINT2000 executables will be compacted by 23% - 29% for the non byte addressed SCISM and 21% - 27% for the byte addressed SCISM.

Conclusions: This paper we have shown how the SCISM tagging can be applied to IA-64 instruction set. A straight-forward and hence very restricted scenario in respect to SCISM was applied. All important IA-64 properties where preserved with a marginal increase in opcode length. On the other hand significant static code size reduction was shown (between 21 and 29 %) due to nops elimination.

References

1. Tomasulo R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal*, vol. 11, pp. 25–33, Jan. 1967.
2. Kemal Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," in *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*. Apr. 1988, pp. 3–21, Elsevier Science Publishers.
3. M. D. Smith, M. Lam, and M. A. Horowitz, "Boosting beyond static scheduling in a superscalar processor," in *Proceedings of the 17th Annual Symposium on Computer Architecture*, 1990, pp. 344–354.
4. Marketing Brochure, *The Series 10000 Personal Supercomputer*, Apollo Computer Inc., Chelmsford, MA, 1988.
5. S. Vassiliadis, B. Blaner, and R. J. Eickmeyer, "SCISM: A scalable compound instruction set machine," *IBM J. Res. Develop.*, vol. 38, no. 1, pp. 59–78, Jan 1994.
6. J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir, "Introducing the IA-64 architecture," *IEEE Micro*, pp. 12–23, Sep-Oct 2000.
7. Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual Vol.3, Rev. 1.0*, 2000.
8. Gerrit Blaauw and Frederick Brooks Jr., *Computer Architecture*, Addison-Wesley, One Jacob Way, 1997.
9. S. Vassiliadis, J.E. Philips, and B. Blaner, "Interlock collapsing ALUs," *IEEE Trans. Computers*, vol. 42, no. 7, pp. 825–839, July 1993.
10. IBM Corporation, *IBM enterprise System Architecture/370 Principles of operation*, 1989.
11. S. Vassiliadis and B. Blaner, "Concepts of the SCISM organization," Technical Report TR-01 C209, IBM Glendale Laboratory, Endicott, NY, Jan 1992.
12. R.J. Eickemeyer, S. Vassiliadis, and B. Blaner, "An in-memory preprocessor for SCISM instruction-level parallel processors," Technical Report TR-01 C407, IBM Glendale Laboratory, Endicott, NY, May 1992.
13. G. N. Gaydadjiev and S. Vassiliadis, "What SCISM tagging can do that IA64 can not," Tech. Rep. CE-TR-2004-02, TU Delft, 2004.
14. Stephane Eranian and David Mosberger, "The making of linux/ia64," *HPL*, Aug. 1999.