Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Kompression von Tracedaten auf Bitebene basierend auf einem LZ77-Wörterbuchansatz

Kai-Uwe Irrgang[1] Thomas B. Preußer[2] Rainer G. Spallek[2]

Kurzfassung: Online-Tracing ist ein leistungsfähiger Ansatz für das Monitoring und Debugging von eingebetteten Prozessoren. Die aufgezeichneten Tracedaten ermöglichen nachfolgend selbst aufwendige Analysen von Systemläufen. Während des Tracings wird ein sehr hohes Datenvolumen erzeugt. Dieses stellt eine große Herausforderung für Speichersysteme und Kommunikationsschnittstellen dar. Eine unmittelbare, möglichst quellnahe und signifikante Datenreduktion in Echtzeit ist außerordentlich wichtig. Die beiden Hauptstrategien dafür sind die Relevanzfilterung und die Kompression der Tracedaten.

Beim Instruktionstrace existieren mehrere effiziente Lösungen für die Kompression von Zieladressen ausgeführter Verzweigungsbefehle. Moderne Befehlssatzarchitekturen erweitern jedoch die Möglichkeit der bedingten Befehlsausführung über Verzweigungsbefehle hinaus auf mehr oder gar alle Befehlsgruppen. Dies führt zu umfangreichen Strömen von Ausführungsbits, die anzeigen, ob die jeweiligen bedingten Instruktionen ausgeführt wurden oder nicht. Dieser Beitrag stellt einen Kompressor für solche Bitströme auf der Basis eines LZ77-Wörterbuchansatzes vor. Im Gegensatz zu klassischen Implementierungen werden Wiederholungen auf Bitebene detektiert. Bereits mit kleinen Wörterbuchlängen wird eine effiziente Kompression erzielt. Dies ermöglicht die Implementierung z.B. in FPGAs. Es wird gezeigt, dass der Kompressor mit der Geschwindigkeit praktisch relevanter Tracedatenquellen arbeiten kann.

Schlüsselwörter: Instruktionstrace, On-Chip-Tracing, Kompression, Ausführungsbits, LZ77, FPGA

1 Einleitung

Ein wesentliches Kennzeichen von in eingebetteten Systemen laufender Software ist deren stetig wachsende Komplexität. Damit wird auch der Software-Test zu einer zunehmend kritischen und zeitaufwendigen Aufgabe. Softwaredebugging und Validierung haben einen großen und weiter steigenden Anteil am gesamten Entwurfsaufwand. Das nicht-invasive Beobachten und Aufzeichnen von Systemzuständen, während das System in Echtzeit und in realer Interaktion mit der Systemumgebung läuft, ist ein wesentlicher Bestandteil des Software-Tests. Dieses passive Abtasten wird als *Tracing* bezeichnet. Typische mittels Tracing realisierte Funktionen und Aufgaben sind die nicht-invasive Analyse des Laufzeitverhaltens, die Fehlerdetektion und die Messung der Code-Abdeckung. Für die Zertifizierung sicherheitskritischer Systeme existieren beispielsweise Forderungen nach Echtzeit-Tracedaten für alle Verzweigungsmöglichkeiten im Code.

[1] Brandenburgische Technische Universität Cottbus-Senftenberg, Fakultät für Ingenieurwissenschaften und Informatik, Institut für Medizintechnologie, Großenhainer Str. 57, 01968 Senftenberg, E-Mail: *irrgang@b-tu.de*
[2] Technische Universität Dresden, Fakultät Informatik, Institut für Technisch Informatik, Nöthnitzer Straße 46, 01187 Dresden, E-Mail: *{thomas.preusser,rainer.spallek}@tu-dresden.de*

Aus der Integration von Systemen in Systems-on-a-Chip (SoCs) ergeben sich neue Herausforderungen und Problemstellungen für Test und Debugging. Das Hauptproblem ist die erheblich eingeschränkte Beobachtbarkeit und Beeinflussbarkeit der eingebetteten Prozessorkerne. Um zumindest teilweise das Niveau von Logikanalysatoren (LA) und In-Circuit-Emulatoren (ICE) zu erreichen, ist die Integration von dedizierter Trace-Hardware in die SoCs unverzichtbar. Der Hardwareaufwand für ein solches On-Chip-Tracing muss so klein wie möglich sein, um die Kosten für zusätzliche Chipfläche zu minimieren.

Eine wesentliche Herausforderung beim On-Chip-Tracing ist das große Volumen der anfallenden Tracedaten. Die Bandbreite des Traceinterfaces und die Größe des Tracepuffers auf dem Chip sind die hauptsächlichen limitierenden Faktoren. Eine signifikante und unmittelbare Volumenreduktion möglichst nahe der Entstehungsquelle ist beim On-Chip-Tracing unverzichtbar. Die beiden Hauptstrategien sind (a) die Relevanzfilterung durch einen Hardware-Tracemonitor, dessen Triggerbedingungen entsprechend der Analyseaufgabe konfiguriert werden und (b) die Kompression der Tracedaten. Die erste Strategie erfordert vertiefte Kenntnisse des Triggersystems und der Analyseaufgaben. Der Anwender muss entsprechend komplexe Konfigurationen ausführen, um die interessanten Intervalle zu erfassen, ohne dabei relevante Daten zu verlieren. Die Kompression hingegen ist ein Ansatz, der keine Benutzerinteraktion erfordert und nicht auf bestimmte Anwendungen zugeschnitten ist. Sie ist damit ein wertvoller generischer Ansatz, der auch im Anschluss an eine vorgelagerte Relevanzfilterung gewinnbringend eingesetzt werden kann.

In einem SoC können prinzipiell alle Komponenten Quellen von Tracedaten sein. In Bezug auf das Tracing von Prozessoren ist der Instruktionstrace der wichtigste. Eine Vielzahl von Analyseaufgaben kann bereits durch dessen alleinige Anwendung ausgeführt werden. Andere Tracedaten, wie beispielsweise ein Datentrace, sind oft nur im Kontext des aus dem Instruktionstrace rekonstruierten Instruktionsflusses bewertbar.

Bei älteren oder weniger leistungsfähigen Befehlssätzen können nur Verzweigungsbefehle bedingt sein. Der Instruktionsfluss kann sich nur an deren Positionen ändern. Alle dazwischen liegenden Befehle werden streng sequentiell ausgeführt. Für die vollständige, auf dem Binärcode basierende Rekonstruktion des Instruktionsflusses genügt also das Tracing der Verzweigungen. Moderne Befehlssätze hingegen erweitern die bedingte Befehlsausführung auf deutlich mehr oder sogar alle Befehlsgruppen. Ein vollständiger Instruktionstrace muss dann die Ausführungsinformation jedes bedingten Befehls enthalten. Der einfachste Ansatz ist das Aufzeichnen von Ausführungsbits, die anzeigen, ob der jeweilige Befehl ausgeführt wurde. Im Fokus dieses Beitrages steht die komprimierte Kodierung von Strömen aus Ausführungsbits. Ein vollständiger Instruktionstrace muss außerdem die nicht aus dem Binärcode ableitbaren Zieladressen indirekter Verzweigungen mitprotokollieren.

Dieser Beitrag ist wie folgt strukturiert. In Abschnitt 2 werden bestehende Architekturen zur Echtzeit-Kompression von Tracedaten sowie allgemeine Hardwareimplementierungen von LZ-Kompressoren vorgestellt. Die vorgeschlagene Kompressorarchitektur mit Diskussion von Implementierungsaspekten wird in Abschnitt 3 erörtert. Das Format der komprimierten Daten wird in Abschnitt 4 definiert. Nach der abschließenden Bewertung des Kompressors in Abschnitt 5 wird der Beitrag in Abschnitt 6 zusammengefasst.

2 Stand der Technik

Für die Kompression von Zieladressen ausgeführter Verzweigungsbefehle existieren mehrere Lösungen, von denen einige sehr hohe Kompressionsraten erzielen können. Uzelac et al. [UM09] nutzen das Double-Move-to-Front-Verfahren (DMTF) zur Detektion von sich wiederholenden Instruktionsströmen. Die Signalisierung von Wiederholungen erfolgt im Idealfall mit nur einem Bit. Jeder Instruktionsstrom besteht aus der Startadresse und der Länge. Milenkovic et al. [MUMB11] nutzen Caches und Prädiktoren anstatt History-Tabellen zur Detektion von Wiederholungen. Kao et al. [KHH07] schlagen einen LZ-basierten Kompressor für Basisblöcke vor. Die Basisblöcke werden hierbei zur Reduktion der Bitbreite und damit des Hardwareaufwandes des Wörterbuches in Slices aufgeteilt. Basisblöcke und Instruktionsströme sind prinzipiell gleich. Die Länge eines Basisblockes ist die arithmetische Differenz aus der Startadresse des nächsten Basisblockes und der eigenen Startadresse und die Länge eines Instruktionsstromes ist gleich der Anzahl der enthaltenen Instruktionen.

Im Gegensatz zur Kompression von Adressen, Instruktionsströmen und Basisblöcken ist die Kompression von Strömen aus Ausführungsbits ein weitestgehend unbehandeltes Feld. Alle vorstehend aufgeführten Lösungen arbeiten ausschließlich mit Adressen. Nur sehr wenige On-Chip-Trace-Architekturen beziehen die Ausführungsbits überhaupt in das Tracing ein. Im History-Mode von Nexus [IEE12] werden die Ausführungsbits in ein Schieberegister eingetaktet und unkomprimiert in paralleler Form ausgegeben. Die Embedded Trace Macrocell (ETM) [ARM11] als Komponente von ARM CoreSight bildet die Ausführungsbits in einem Bytestrom ab. Hierbei existieren zwei Typen von Einzelbytes, die als P-Header bezeichneten werden. Ein P-Header im Format 2 enthält genau zwei unkomprimierte Ausführungsbits. In einem P-Header im Format 1 sind maximal 16 Ausführungsbits in zwei Binärfeldern kodiert: ein 4-Bit-Zähler repräsentiert null bis 15 konsekutive Befehle, deren Ausführungsbedingung *wahr* war, und ein 1-Bit-Flag zeigt an, ob diesen ein oder kein Befehl mit der Ausführungsbedingung *falsch* folgte.

Die größten Anteile am Volumen von Instruktionstracedaten moderner Prozessoren und Befehlssatzarchitekturen haben (a) Zieladressen ausgeführter Verzweigungen und (b) Ströme aus Ausführungsbits. Zieladressen von direkten Sprüngen können aus dem Binärcode abgelesen werden, so dass deren Aufzeichnung verzichtbar ist. Die Ausführung bedingter direkter Sprünge wird gemeinsam mit allen anderen bedingten Befehlen in den Bitsequenzen aufgezeichnet. Unbedingte direkte Sprünge werden wie sequentielle Befehle behandelt. Mit diesen Festlegungen liegt das Rohdatenvolumen der Ausführungsbits im Median um rund den Faktor 14 über dem der Adressen. Die Streuung über die Programme ist sehr groß. Das Maximum dieses Faktors beträgt rund 22000. Andererseits existieren aber auch Programme, bei denen das Volumen des Adressanteils um den Faktor 128 über dem der Ausführungsbits liegt.

Damit ein Prozessor die benötigten Rohdaten, die Zieladressen und die Ausführungsbits, der Trace-Hardware bereitstellen kann, sollte vorzugsweise dessen Befehlsdecoder so erweitert werden, dass alle tatsächlich zur Ausführung gelangten Befehle diese Daten mit dem Befehlstakt signalisieren.

Das wörterbuchbasierte verlustlose Datenkompressionsverfahren LZ77 ist in vielen verschiedenen Anwendungsgebieten verbreitet. Es ist Bestandteil von DEFLATE, welches ursprünglich von Phil Katz [Deu96b] entwickelt wurde. DEFLATE kombiniert LZ77 mit einer nachfolgenden HUFFMAN-Codierung. Dieser Ansatz wurde auch von Gailly und Adler für deren Dateikompressor GZIP [Deu96c] und für die ZLIB Bibliothek [Deu96a] adaptiert. Neben diesen weit verbreiteten Standardwerkzeugen sind DEFLATE und dessen Derivate allgegenwärtig in der Internetkommunikation bei der verlustlosen Kompression von Grafiken im PNG-Format, des HTTP-Datentransfers oder von SSH-Verbindungen.

Für die Implementierung von LZ77-basierten Kompressoren direkt in Hardware existieren mehrere Vorschläge. Grajeda et al. [GUCP06] stellen einen Hardwarebeschleuniger für die LZ77-Kompression vor, der an einen LEON2-Prozessor angebunden ist. Der Beschleuniger kann zusätzlich eine Burrows-Wheeler-Transformation zur Vorverarbeitung der Daten ausführen. Angewendet wird diese Architektur für eine unmittelbare Online-Datenkompression in vernetzten Kommunikationsgeräten. Rigler et al. [RBK07] beschreiben ein FPGA-basiertes System, bei dem eine LZ77-Kompression und eine HUFFMAN-Codierung in Hardware implementiert sind. Damit sind wesentliche Komponenten für die Hardware-Implementierung von GZIP geschaffen. Ältere Vorschläge für LZ-Hardware-Kompressoren stammen von Chen und Wei [CW99] sowie von Huang et al. [HSM00]. Lin beschreibt eine Hardware-Architektur des LZW-Algorithmus [Lin00]. Alle diese Arbeiten zielen nicht speziell auf die Kompression von Tracedaten ab. Die Arbeit von Huang et al. bildet jedoch den Ausgangspunkt für den Basisblock-Kompressor von Kao et al.

Sowohl Software- als auch Hardwareimplementierungen von LZ77 gehen typischerweise davon aus, auf einem Alphabet aus 256 Symbolen zu arbeiten, die durch 8-Bit-Worte darstellbar sind. Das ist eine sinnvolle Wahl für Daten, die in mindestens ein Byte großen Feldern organisiert sind. Für die überwiegende Mehrzahl der Anwendungsszenarien trifft dies zu, es gibt jedoch auch Ausnahmen. Wolff und Papachristou [WP02] schlagen die Anwendung der LZ77-Kompression auf Bitstrings vor, welche die Testmuster für das automatisierte Testen von VLSI-Chips enthalten. Für diesen speziellen Anwendungsfall schlagen sie weiterhin vor, den hohen Anteil von *Don't Cares* auszunutzen, um die erzielbare Kompressionsrate weiter zu steigern. Die Kompression wird offline ausgeführt und erfordert zwei Läufe über die Daten. Der erste Durchlauf ersetzt die *Don't Cares* mit festen Werten dergestalt, dass bei der Kompression möglichst lange Bitketten im Wörterbuch gefunden werden. Im zweiten Durchlauf erfolgt die eigentliche Kompression. Nach der Übertragung der komprimierten Testmuster in den Chip werden diese in der Hardware dekomprimiert.

Zusammenfassend ist festzustellen, dass keine oder nur sehr einfache Ansätze zur On-Chip-Kompression von Ausführungsbits in Echtzeit existieren. Der Einsatz von Adress- oder anderen Kompressoren, die mit Bytes oder noch breiteren Datenworten arbeiten, ist hierfür nicht praktikabel. Benachbarte Wiederholungen innerhalb eines Bitstromes sind auf der Ebene von Bytes oder höher in der Regel nicht mehr erkennbar. Eine effektive Kompression würde demnach ein größeres Wörterbuch erfordern oder könnte so nicht realisierbar sein. Die Implementierung eines LZ77-Kompressors, der auf Bitebene in Echtzeit arbeitet, scheint ein vielversprechender Ansatz für die Ausnutzung des bisher unausgeschöpften Kompressionspotenzials zu sein.

3 Architektur des Kompressors

Grundsätzlich wäre es denkbar, dass die Ausführung jedes einzelnen Befehls bedingt ist und damit auch ein Ausführungsbit generiert. In diesem ärgsten Grenzfall müsste der Kompressor in jedem Befehlszyklus ein Bit verarbeiten. Praktisch beobachtet werden deutlich geringere Anteile an bedingten Instruktionen. Bei den untersuchten EEMBC-Benchmarks lag deren maximaler Anteil bei 45,8%. Dies erfordert immer noch die Fähigkeit, alle zwei Befehlszyklen ein Bit zu verarbeiten. Der durchschnittliche Anteil bedingter Befehle über alle Benchmark liegt jedoch mit 14,5% noch einmal deutlich unter diesem Wert.

Um einen uneingeschränkten praktischen Einsatz mit einem möglichst vollständigen Trace zu gewährleisten, sollte ein Kompressor, der in jedem Takt ein Ausführungsbit verarbeiten kann, zumindest mit der halben Taktfrequenz des beobachteten Prozessors laufen. Lokale Bursts werden mit einem FIFO von mindestens 10 Bit vollständig ausgeglichen. Taktfrequenzen um 50 MHz werden also in der Regel genügen, wenn die Trace-Quelle ein auf demselben FPGA integrierter Soft-Core-Prozessor ist. Deutlich schneller taktende festverdrahtete externe oder integrierte Prozessoren verlangen schon nach 200 MHz oder mehr. Letztendlich könnte auch ein festverdrahteter Kompressor notwendig sein, um Kerne mit Taktfrequenzen im GHz-Bereich zu bedienen.

Die erzielbare Taktfrequenz hängt stark von der Größe des verwendeten Wörterbuches ab. Ein großes Wörterbuch kommt dem erreichbaren Kompressionsgrad dadurch zugute, dass das Auffinden von Wiederholungen in einem größeren Korpus wahrscheinlicher ist. Die Notwendigkeit der unmittelbaren weiteren Verarbeitung der Eingabedaten schränkt jedoch die Komplexität der realisierbaren Suche ein. Während eine Hardware-Implementierung durchaus einen parallelen Vergleich über alle Wörterbuchpositionen erlaubt, muss aber immer noch die Existenz eines Treffers über eine Berechnung mit einem großen Fanin bestimmt werden. Diese dominiert schnell den kritischen Pfad der Gesamtimplementierung.

Ein weiterer Grund das Wörterbuch eher klein zu halten, liegt im Ressourcenbedarf des *History Buffers*. Während der Ressourcenbedarf der Steuerlogik weitestgehend unabhängig von der Auslegung des Kompressors ist, wächst beim *History Buffer* der Bedarf an Registern und Vergleicherlogik im Gleichschritt mit seiner Größe. Das Ausweichen auf kompaktere RAM-Strukturen ist nicht möglich, da diese den für die parallelen Vergleiche notwendigen gleichzeitigen Zugriff auf den Speicherinhalt nicht erlauben. Ein alternativer Ansatz über eine Hash-Tabelle würde zwar recht schnell die Position eines möglichen Treffers identifizieren, dessen Verifikation müsste daraufhin jedoch durch einen iterativen Vergleich von Speicherbereichen erfolgen. Der Implementierung des *History Buffers* in Registern, die eng mit den parallelen Vergleichern gekoppelt sind, wurde deshalb der Vorzug gegeben.

Bei einer Implementierung in einem FPGA konkurriert die Verwendung von allgemeinen Registern stark mit dem Ressourcenbedarf des überwachten Prozessorsystems. Eine möglichst geringe Beschränkung des Cores und seiner Erweiterungen verlangt nach einem kleinen *History Buffer*. Auch bei einer Realisierung innerhalb der Trace-Architektur eines hartverdrahteten Prozessors beansprucht der *History Buffer* wertvolle Chipfläche, die mit seiner Größe wächst und im nicht überwachten Normalbetrieb ungenutzt bleibt.

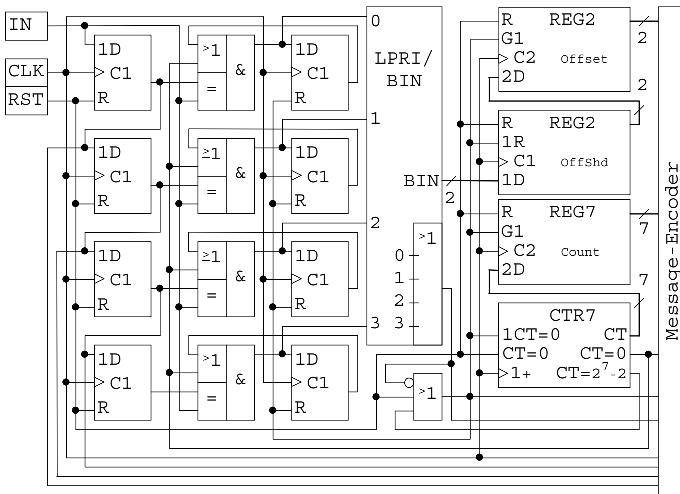

Abbildung 1: Vereinfachtes Schaltbild des Kompressors für Ausführungsbits

Abb. 1 zeigt den Aufbau des Kompressors. Die linke Registerspalte realisiert den *History Buffer*, dessen Inhalt das verfügbare Wörterbuch bereitstellt, als Schieberegister. Die dazu parallele Registerspalte speichert den aktuellen Vergleichszustand für jede Wörterbuchposition. Anfangs zeigen alle Vergleicher eine Übereinstimmung mit dem leeren Wort an. Jede weitere Eingabe eines Ausführungsbits wird mit dem Wert des aktuell im *History Buffer* befindlichen Bits an dieser Stelle verglichen und setzt die Übereinstimmung so fort oder beendet sie. Der Vergleichszustand zeigt also an, ob die seit dem letzten Start eingegebene Bitsequenz mit der an einer Position vorbeigeschobenen übereinstimmt. Beendet eine Eingabe die letzte verbliebene und damit längste Übereinstimmung oder endet die Eingabe gänzlich, so wird diese Übereinstimmung unter Angabe von Position und Länge ausgegeben. Ist sie kurz, so erfolgt die Ausgabe jedoch effizienter als Literal, also in Form einer Kopie der zuletzt konsumierten Bitfolge.

4 Komprimiertes Datenstromformat

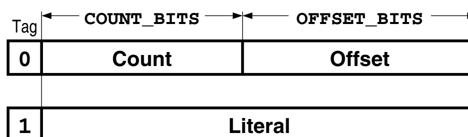

Abbildung 2: Format eines Eintrags im komprimierten Datenstrom

Der komprimierte Datenstrom wird aus Einträgen fester Länge gebildet, deren Format Abb. 2 zeigt. Es gibt zwei Typen von Einträgen: (a) Literale und (b) Rückverweise zur Kodierung von Wiederholungen. Das höchstwertige Bit eines Eintrages, sein *Tag*, differenziert diese beiden Typen. Die Bitlänge eines einzelnen Eintrages wird durch die beiden Codeparameter `COUNT_BITS` und `OFFSET_BITS` bestimmt.

Ein gesetztes Tag-Bit identifiziert ein Literal. Dessen restliche Bits sind zur Dekompression von der höchstwertigen bis zur niederwertigsten Stelle in unveränderter Weise in die Ausgabe zu kopieren. Ein Literal kodiert also eine *unkomprimierte* Bitsequenz der Länge `COUNT_BITS+OFFSET_BITS`.

Ein gelöschtes Tag-Bit identifiziert (in der Regel) einen Wiederholungseintrag. Das Feld Offset gibt an, an welcher Position im *History Buffer* das jetzt wiederkehrende Muster bereits zu finden ist, wobei dem zuletzt ausgegebenen Bit Null und älteren Ausgaben entsprechend höhere Offsets zugewiesen sind. Die Wahl des Codeparameters `OFFSET_BITS` bestimmt den Wertebereich dieser Positionsangabe und damit die Größe des Wörterbuches, das der Kompression und Dekompression zugrunde liegt. Die Länge der Wiederholung wird durch das Feld Count spezifiziert. Nicht unterstützt werden hierbei Wiederholungen von weniger als `COUNT_BITS+OFFSET_BITS` Bits, da die betroffenen Bitsequenzen effizienter in einem Literal kodiert werden können. Um keinen Coderaum zu verschenken, ist die tatsächliche Wiederholungslänge wie folgt zu berechnen:

$$\texttt{COUNT_BITS} + \texttt{OFFSET_BITS} + \text{Count}$$

Typische LZ77-basierte Formate würden nach einem Wiederholungseintrag das nächste, nicht mehr übereinstimmende Zeichen implizit als eine Art Literal kodieren. Bei der Verarbeitung von Bitsequenzen kann der Wert dieses ersten nicht mehr übereinstimmenden Zeichens jedoch als Negation des im Wörterbuch auf die Wiederholung folgenden Bits inferiert werden, so dass dieses Bit gar nicht kodiert werden muss.

Die beiden höchsten möglichen Werte des Count-Feldes werden zur Kodierung von Spezialfällen genutzt. Falls $\text{Count} = 2^{\texttt{COUNT_BITS}} - 2$, so entfällt das implizit folgende nicht mehr übereinstimmende Bit. Dies erlaubt die stückweise Kodierung einer Wiederholung, deren Länge die vorgesehene Kapazität des Count-Feldes sprengen würde.

Der Wert $\text{Count} = 2^{\texttt{COUNT_BITS}} - 1$ markiert schließlich das Ende des Bitstromes. Das Offset-Feld wird in diesem Falle genutzt, um die Werte der letzten Bits zu korrigieren. Ist der Offset 0 oder 1, so gibt sein niederwertigstes Bit den tatsächlichen Wert des letzten Bits an. Solche Endmarkierungen finden sich hinter einem abschließenden Wiederholungseintrag, der ja nicht notwendigerweise durch eine fehlende Übereinstimmung beendet wurde, sondern möglicherweise schlicht durch das Ende der Eingabe. Andere Werte im Offset-Feld kodieren die Anzahl der Bits, um die das vorangegangene letzte Literal zu lang war, im Einerkomplement. Diese Korrektur ist notwendig, da es ansonsten keine Möglichkeit gibt, Literale kürzer als $\texttt{COUNT_BITS} + \texttt{OFFSET_BITS}$ darzustellen. Die relevanten Bits des letzten Literals befinden sich in den niederwertigen Bits, also rechts ausgerichtet.

Aus der beschriebenen Kodierung ergeben sich einige Randbedingungen für die Codeparameter. Um überhaupt eine Kompression zu ermöglichen, muss $\texttt{COUNT_BITS} > 1$ sein. Für den Fall $\texttt{COUNT_BITS} = 1$ könnten nur die beiden Spezialfälle, die Endmarkierung und der Überlauf von Count ohne implizit folgendes nicht mehr übereinstimmendes Bit, kodiert werden. Beide Typen von Einträgen würden so zur Kodierung von $\texttt{COUNT_BITS} + \texttt{OFFSET_BITS}$ Bits sogar ein zusätzliches Bit benötigen.

In einer tatsächlichen Realisierung sollte die Wahl von COUNT_BITS die Kodierung typischer Wiederholungslängen ermöglichen. Nimmt man die Implementierung von DEFLATE als vagen Anhaltspunkt [Deu96b, § 3.2.5], so erscheinen Feldgrößen von etwa 8 Bit sinnvoll. Intuitiv sollte auch die Länge des verfügbaren Wörterbuches nicht kürzer als diese darstellbare Länge sein, so dass $\texttt{COUNT_BITS} \leq \texttt{OFFSET_BITS}$. Dazu ist jedoch anzumerken, dass das Format (und der implementierte Kompressor) die Kodierung von in sich selbst wiederholenden Bitsequenzen erlaubt. Bei diesen ist die Wiederholungslänge größer als das Offset in das Wörterbuch, so dass die Wiederholung in einen Bereich des Wörterbuches hineinreicht, der erst durch diese Wiederholung definiert wird. Tatsächlich wird ein anfänglicher nur aus Nullen bestehender Wörterbuchinhalt angenommen[1], so dass ein mit n Nullen beginnender Bitstrom als erstes eine Wiederholungsnachricht der Länge n an Offset 0 produziert.

Um die korrekte Kodierung aller Endmarkierung zu ermöglichen, muss das Offset-Feld jede mögliche Zahl von überflüssigen Bits im letzten Literal darstellen können, ohne dass die besonderen Offset-Werte 0 und 1 zur Anwendung kommen. Das heißt:

$$(\texttt{COUNT_BITS} + \texttt{OFFSET_BITS} - 1) + 2 \leq 2^{\texttt{OFFSET_BITS}}$$

$$\texttt{COUNT_BITS} + \texttt{OFFSET_BITS} < 2^{\texttt{OFFSET_BITS}}$$

Die Wahl der Codeparameter bestimmt feste Schranken für die erreichbare Kompressionsrate. Eine stark zufällige Eingabe, in der keine Wiederholung im Bereich des verfügbaren Wörterbuches genutzt werden können, würde als reine Folge aus Literalen kodiert. Dies würde das Datenvolumen um den folgenden Faktor aufblähen:

$$\frac{\texttt{COUNT_BITS} + \texttt{OFFSET_BITS} + 1}{\texttt{COUNT_BITS} + \texttt{OFFSET_BITS}}$$

Die beste Kompression wird mit Wiederholungseinträgen maximaler Länge erreicht. Solche Einträge können ein Count-Feld mit dem Wert $2^{\texttt{COUNT_BITS}} - 3$ haben, so dass auf die Wiederholung ein implizites nicht mehr übereinstimmendes Bit folgt; oder sie haben ein Count-Feld mit dem speziellen Wert $2^{\texttt{COUNT_BITS}} - 2$, bei dem dieses implizite Bit entfällt. In beiden Fällen ergibt sich die erreichbare Kompressionsrate zu:

$$\frac{\texttt{COUNT_BITS} + \texttt{OFFSET_BITS} + 1}{2^{\texttt{COUNT_BITS}} - 2}$$

5 Ergebnisse

Aufgrund der weiten Verbreitung und Typenvielfalt von ARM-Prozessoren werden die Untersuchungen für aktuelle Cortex-Prozessoren durchgeführt. Die Familie der ARM-Prozessoren unterstützt drei Befehlssätze, die sich bezüglich der bedingten Befehlsausführung unterscheiden. Beim Thumb-Befehlssatz des Cortex-M0 können nur Verzweigungsbefehle bedingt sein. Beim erweiterten Thumb2-Befehlssatz des Cortex-M4 existiert der

[1] Das definierte Vorhalten anderer Bitmuster im initialen Wörterbuch könnte die anfängliche Kompressionsrate verbessern, hat aber keine Auswirkungen auf eine eingelaufene Kompression.

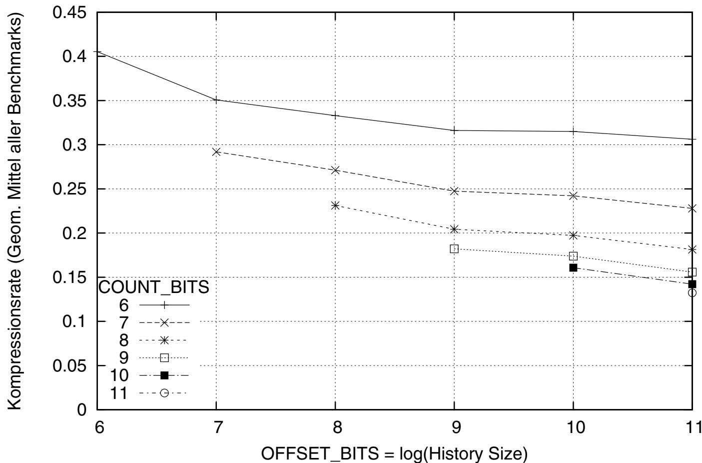

Abbildung 3: Kompressionsrate in Abhängigkeit von den Code-Parametern

Befehl `IT`, der einen If-Then-Else-Block über die unmittelbar nachfolgenden ein bis vier Befehle aufspannt. Die Ausführung dieser Befehle hängt von einer gemeinsamen Bedingung ab, die mit dem `IT`-Befehl angegeben wird und für die mit `if` markierten Befehle gilt. Die Ausführungsbedingung der mit `else` markierten Befehle ist implizit deren Negation. Der ARM-Befehlssatz ist noch flexibler und erlaubt für nahezu jeden Befehl eine eigene, von den anderen unabhängige Ausführungsbedingung. Um sicherzustellen, dass alle Familien und Befehlssätze einbezogen werden, basieren die Untersuchungen auf:

- dem Cortex-M0 und Cortex-M4(F) aus der Familie der Mikrocontroller,
- dem Cortex-A5 und Cortex-A5N aus der Familie der Applikations-Prozessoren und
- dem Cortex-R4F und Cortex-R7F aus der Familie der Realtime-Prozessoren.

Der Einfluss einer zusätzlichen Gleitkommaeinheit ist aus dem Vergleich zwischen dem Cortex-M4F mit FPU und dem Cortex-M4 ohne FPU sowie aus dem Vergleich zwischen dem Cortex-A5N mit der Vektoreinheit NEON und dem Cortex-A5 ersichtlich.

Als einheitliche Basis für die vergleichende Bewertung aller Kompressions-Architekturen und -Methoden ist die EEMBC Automotive Benchmark Software [EEM07] ausgewählt worden. Diese umfasst 16 repräsentative und typischerweise von eingebetteten Systemen ausgeführte Benchmarkprogramme. Zwei Benchmarks sind ausgewiesene Gleitkomma-Benchmarks und weitere zwei enthalten Gleitkommaoperationen. Die Anzahl der jeweils ausgeführten Iterationen übersteigt den Wertevorrat verschiedener Eingangswerte in keinem Fall. Damit ist die Wiedererkennung und Kompression ganzer Iterationen infolge identischer Eingangsdaten ausgeschlossen. Jedes Programm ist mit den Compileroptimierungen *None*, *Size* und *Speed* compiliert worden. Genutzt wurde der Compiler von IAR mit den Optionen *-On*, *-Ohs* und *-Ohz*.

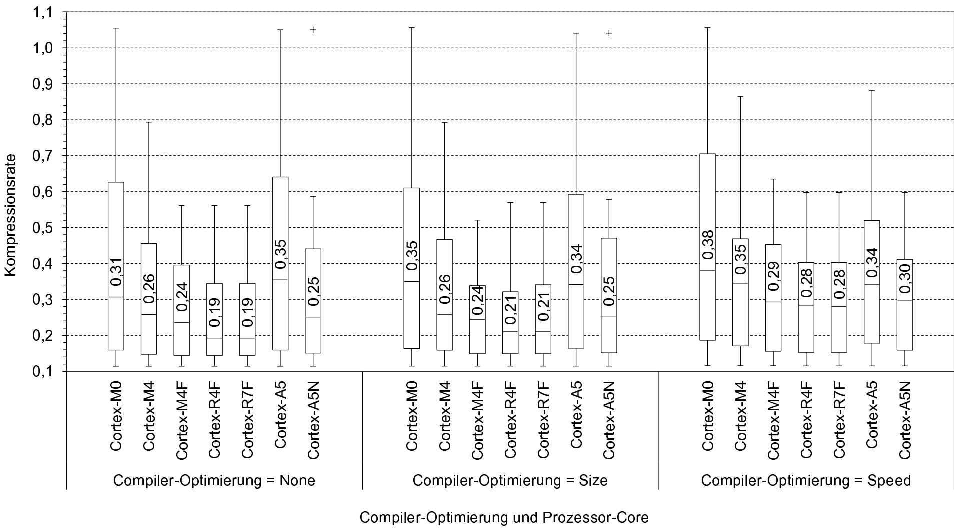

Abbildung 4: Kompressionsrate des Kompressors für Ausführungsbits

Die Abb. 3 zeigt den Einfluss der Code-Parameter auf die erzielbare Kompressionsrate. Innerhalb des betrachteten Parameterraumes erhöhen sowohl eine Vergrößerung von `COUNT_BITS` als auch die von `OFFSET_BITS` die Kompressionsrate. Beide Parameter beeinflussen den Aufwand für die Codierung der Wiederholungseinträge. Der Parameter `OFFSET_BITS` determiniert zudem die Länge des Wörterbuches. Infolge des Codieraufwandes ist ein Absinken der Kompressionsrate mit weiter wachsenden Feldbreiten wahrscheinlich. Alle nachfolgenden Evaluationen basieren auf einer Parametrierung mit einem relativ kleinen Wörterbuch von 256 Bits (`OFFSET_BITS` = 8) und einer maximalen Wiederholungslänge von 126 Bits (`COUNT_BITS` = 7).

Aus der Abb. 4 sind die damit erzielten Kompressionsraten als Tukey Boxplots ablesbar. Eine solche Darstellung ermöglicht eine detailliertere Sicht auf die Daten und deren Verteilung als die alleinige Angabe eines Mittelwerts. Insbesondere das oft verwendete arithmetische Mittel ist nicht robust genug und daher ungeeignet. Alle Kombinationen aus Prozessoren und Compileroptimierungen sind einbezogen. Somit umfasst jeder Boxplot 16 Benchmarkprogramme.

In der gewählten Konfiguration mit `COUNT_BITS` = 7 und `OFFSET_BITS` = 8 beträgt die maximal erzielbare Kompression rund 0,113. Das entspricht einer Reduktion des Datenvolumens auf 11,3% der ursprünglichen Größe. In allen Kombinationen wird dieses Maximum von mindestens einem Benchmark nahezu erreicht. Die minimale Kompression beträgt rund 1,067. Das Datenvolumen wird in diesem Fall um 6,7% erhöht. Dieses Minimum tritt nur in seltenen Fällen beim Cortex-M0 und Cortex-A5 auf. In der überwiegenden Mehrzahl von 98% aller Benchmarks wird eine Reduktion des Datenvolumen erreicht. Im Median wird das Volumen um den Faktor drei bis vier reduziert. Für 80% aller Benchmarks wird das Volumen mindestens halbiert. Der Einfluss der Compileropti-

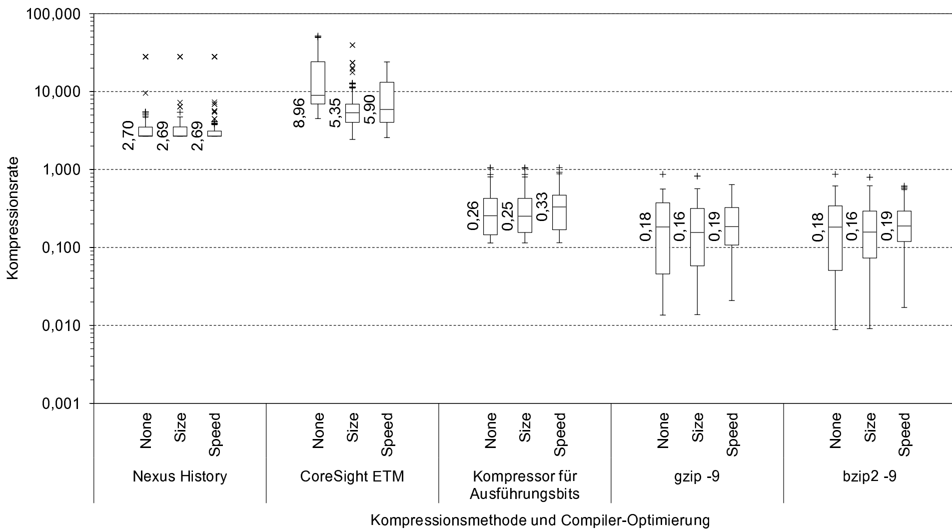

Abbildung 5: Vergleich der Kompressionsraten des Kompressors für Ausführungsbits mit anderen Architekturen und Methoden

mierung ist vergleichsweise klein. Die besten mittleren Kompressionsraten werden bei *None* erzielt. Bei *Size* verschlechtern sich die Ergebnisse nur geringfügig. Für *Speed* ist ein etwas stärkerer Rückgang festzustellen.

Der Prozessorkern hat einen größeren Einfluss auf die Kompressionsrate als die Compileroptimierung. Die schlechtesten Ergebnisse werden mit dem Cortex-M0 erzielt. Allerdings ist dieser auch der einzige Prozessor, bei dem nur Verzweigungsbefehle bedingt sein können. Mit dem Cortex-A5 werden nur geringfügig bessere Ergebnisse erreicht. Durch die Hinzunahme der Vektoreinheit NEON (Cortex-A5N) wird die Kompressionsrate deutlich verbessert. Dieser positive Einfluss einer FPU auf die Kompression ist auch aus dem Vergleich zwischen dem Cortex-M4 und dem Cortex-M4F erkennbar. Die besten Ergebnisse werden mit den Realtime-Prozessoren Cortex-R4F und Cortex-R7F erzielt.

Das Hauptmotiv für die durchgeführten Untersuchungen war die Frage, ob eine nennenswerte Kompression von Ausführungsbits innerhalb des Instruktionstrace von modernen Prozessoren mit moderatem Hardwareaufwand und in Echtzeit möglich ist. Den Ausgangspunkt für die Vergleiche bilden die einzigen beiden On-Chip-Trace-Architekturen, Nexus History und CoreSight ETM, welche die Ausführungsbits überhaupt in den Instruktionstrace einbeziehen. Die Ergebnisse sind in Abb. 5 dargestellt.

Nexus History gibt die Ausführungsbits generell unkomprimiert aus. Aufgrund des Messageformates wird ein Overhead hinzugefügt, so dass sich das Datenvolumen immer erhöht. Im Median wächst das Volumen um den Faktor 2,7. Die von CoreSight ETM erreichten Kompressionsraten sind noch schlechter als die von Nexus History. Relativierend muss hier die abweichende Funktionsweise in Betracht gezogen werden. Die ETM zeichnet nicht nur für jeden bedingten Befehl ein Ausführungsbit auf, sondern auch für jeden un-

Tabelle 1: Ressourcenbedarf des Kompressors für Ausführungsbits

Device	**Register**	**LUTs**	**Taktrate**
XC5VLX50T-FF1136-2 (speed)	550	1635	218 MHz
XC5VLX50T-FF1136-2 (size)	550	1428	107 MHz

bedingten. Ein unbedingter und somit immer ausgeführter Befehl wird in derselben Weise wie ein ausgeführter bedingter Befehl behandelt. Dies führt zu einer Verringerung der Kompressionsrate, wenn auch hier nur die bedingten Befehle als Berechnungsbasis zugrunde gelegt werden. Aber auch dann, wenn dieser Fakt vernachlässigt wird, sind keine wesentlich besseren Resultate als bei Nexus zu erwarten.

Die Vergleiche mit den beiden etablierten On-Chip-Trace-Architekturen verdeutlichen das Kompressionspotenzial in den Ausführungsbits und demonstrieren die Leistungsfähigkeit des hier vorgeschlagenen Kompressors. Gegenüber Nexus History wird das Datenvolumen um einen Faktor von acht oder mehr reduziert.

Abschließend erfolgt der Vergleich der Kompressionsraten mit den für die Kompression von Dateien etablierten Kompressoren bzip2 und gzip. Obwohl der Aufwand für deren vollständige Implementierung in Hardware sehr hoch und oftmals unakzeptabel ist, sind sie dessen ungeachtet eine Referenz für die praktisch erreichbaren Kompressionsraten. Bei beiden Kompressoren wird der Kompressionsaufwand mittels eines Parameters auf der Kommandozeile übergeben. Aufgrund des geringen Einflusses dieses Parameters bei den hier verwendeten Daten ist nur die höchste Aufwandsstufe (`-9`) dargestellt.

Wie erwartet erreichen bzip2 und gzip bessere Kompressionsraten als der hier vorgestellte Kompressor. Diese beiden erzielen ihre Ergebnisse jedoch aufgrund von zusätzlichen Verarbeitungsschritten wie der Huffman-Codierung und Burrows-Wheeler-Transformation sowie aus viel größeren Speicherblöcken für die History. Während gzip mit 32 kB großen Blöcken arbeitet, nutzt bzip2 in Abhängigkeit von der gewählten Aufwandsstufe sogar 100 bis 900 kB große Blöcke. Das liegt um Größenordnungen über der Wörterbuchgröße von 256 Bits (64 Bytes), die in der hier vorgeschlagenen Implementierung verwendet wird. Trotzdem liegt die damit erzielte mittlere Kompression bemerkenswert nahe an denen von bzip2 und gzip. Die Worst-cases sind nur unwesentlich schlechter. Lediglich die Tracedaten, die bereits vom Hardware-Kompressor besonders gut komprimiert werden, werden von bzip2 oder gzip noch einmal deutlich besser verdichtet.

Die Tab. 1 zeigt die Eigenschaften zweier konkreter Hardwareimplementierungen auf einem Virtex-5 FPGA. Für die Ermittlung dieser Werte wurde das Kompressordesign zwischen den Eingangsregistern und den registerbasierten Ausgängen eingebettet, so dass das ISE Synthesewerkzeug die erzielbare Taktfrequenz zuverlässig aus dem kritischen Register-Register-Pfad bestimmen kann. Die angezeigte Ressourcennutzung beinhaltet das komplette Toplevel-Design inklusive der Eingangs- und Ausgangsregister.

Bewertet wurden zwei alternative, funktionell äquivalente Implementierungen. Obwohl beide im generischen, geräte- und herstellerunabhängigen VHDL umgesetzt wurden, ver-

wenden sie verschiedene Ansätze zur Ermittlung des Match-Offsets im Wörterbuch. Diese Berechnung erfordert die Lokalisierung eines gesetzt verbliebenen Match-Signals aus allen 256 Positionen des Wörterbuches. In der geschwindigkeitsoptimierten Variante erzeugt das Synthesewerkzeug einen Reduktionsbaum. Die größenoptimierte Variante nutzt einen Addierer für die Berechnung des Zweierkomplements $-m$ des Match-Vektors m. Aus diesem wird anschließend die bitweise `und`-Verknüpfung `m&-m` gebildet. In deren Ergebnis ist nur das in m äußerst rechts stehende gesetzte Bit gesetzt. Das Umcodieren dieser One-Hot-Codierung in eine binäre Position ist einfacher, so dass insgesamt eine Reduktion um mehr als 200 LUTs erreicht werden kann. Allerdings ist die Carry-Chain des 256-Bit-Addierers bereits recht lang und komplex, so dass die maximale Taktfrequenz erheblich beeinträchtigt wird. Trotzdem ist der Kompressor ausreichend schnell für viele Soft-Core-Prozessoren. Anspruchsvollere Tracequellen wie Hard-Core-Prozessoren erfordern die geschwindigkeitsoptimierte Variante oder gar ein Härten des Kompressors.

Die Ergebnisse zeigen, dass ein auf Bitebene arbeitender LZ77-Kompressor mit einem vergleichsweise geringen Hardwareaufwand eine deutliche Kompression der Tracedaten bewirken kann. Die gewählte Konfiguration mit einer Wörterbuchgröße von 256 Bits ist ein guter Kompromiss zwischen Hardwareaufwand und erzielbarer Kompressionsrate.

6 Zusammenfassung

Tracing als nicht-invasives Echtzeit-Monitoring von Systemzuständen ist für den Software-Test in eingebetteten Systemen unverzichtbar. Die Aufzeichung des Instruktionsflusses ist dabei am wichtigsten. Für komplexe SoCs ist die Integration von Trace-Hardware unentbehrlich. Der Aufwand für ein solches On-Chip-Tracing muss möglichst gering sein. Eine der wesentlichsten Herausforderungen ist die signifikante Reduktion des Tracedatenvolumens. Die Kompression ist eine besonders geeignete Strategie dafür.

Innerhalb des Instruktionstraces von modernen Befehlssatzarchitekturen besteht der größte Teil des Datenvolumens aus zwei verschiedenen Komponenten: Zieladressen indirekter Verzweigungen und Ausführungsbits, die infolge der bedingten Ausführung anderer Befehle entstehen. Während für die Kompression von Adressen mehrere Lösungen existieren, werden Ausführungsbits bisher nur rudimentär oder gar nicht komprimiert. Der vorliegende Beitrag schließt diese Lücke und stellt einen auf Bitebene arbeitenden LZ77-Kompressor für Sequenzen von Ausführungsbits vor. Dessen Implementierung und das Ausgabeformat sind beschrieben worden.

Der vorgestellte Kompressor wurde auf der Basis der EEMBC Automotive Benchmark Suite evaluiert. Jedes der 16 Benchmarkprogramme wurde mit drei Optimierungsleveln compiliert und auf sieben verschiedenen ARM Cortex Prozessoren ausgeführt. Alle drei Familien und alle drei Befehlssätze wurden auf diese Weise in die Untersuchungen einbezogen. Bei 98% aller Benchmarks wird das Datenvolumen reduziert und bei 80% wird es mindestens halbiert. Der Median der Volumenreduktion liegt im Bereich von einem Drittel bis zu einem Viertel. Diese Resultate liegen beachtlich nahe an den von bzip2 und gzip erzielten. Die Implementierung auf einem Virtex-5 FPGA benötigt rund 550 Register, 1500 LUTs und erreicht eine Geschwindigkeit von bis zu 218 MHz.

Literatur

[ARM11] ARM Limited. Embedded trace macrocell architecture specification ETMv1.0 to ETMv3.5, 2011.

[CW99] Jun-Min Chen and Che-Ho Wei. VLSI design for high-speed LZ-based data compression. *IEE Proceedings - Circuits, Devices and Systems*, 146(5):268–278, Oct 1999.

[Deu96a] P. Deutsch. RFC 1950: ZLIB compressed data format specification version 3.3, May 1996.

[Deu96b] P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, May 1996.

[Deu96c] P. Deutsch. RFC 1952: GZIP file format specification version 4.3, May 1996.

[EEM07] EEMBC. EEMBC autobench 1.1 benchmark software, 2007.

[GUCP06] Virgilio Zuñiga Grajeda, Claudia Feregrino Uribe, and René Cumplido-Parra. Parallel hardware/software architecture for the BWT and LZ77 lossless data compression algorithms. *Computación y Sistemas*, 10(2):172–188, 2006.

[HSM00] Wei-Je Huang, Nirmal Saxena, and Edward J. McCluskey. A reliable LZ data compressor on reconfigurable coprocessors. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '00, pages 249–258. IEEE Computer Society, Apr 2000.

[IEE12] IEEE-ISTO. The nexus 5001 forum standard for a global embedded processor debug interface version 2.0 (IEEE-ISTO 5001-2003), 2012.

[KHH07] Chung-Fu Kao, Shyh-Ming Huang, and Ing-Jer Huang. A hardware approach to real-time program trace compression for embedded processors. *IEEE Trans. on Circuits and Systems I: Regular Papers*, 54(3):530–543, Mar 2007.

[Lin00] Ming-Bo Lin. A hardware architecture for the LZW compression and decompression algorithms based on parallel dictionaries. *J. VLSI Signal Process. Syst.*, 26(3):369–381, Nov 2000.

[MUMB11] A. Milenkovic, V. Uzelac, M. Milenkovic, and M. Burtscher. Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems. *Computers, IEEE Transactions on*, 60(7):992–1005, July 2011.

[RBK07] S. Rigler, W. Bishop, and A. Kennings. FPGA-based lossless data compression using Huffman and LZ77 algorithms. In *Canadian Conference on Electrical and Computer Engineering (CCECE 2007)*, pages 1235–1238, Apr 2007.

[UM09] V. Uzelac and A. Milenkovic. A real-time program trace compressor utilizing double move-to-front method. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 738–743, July 2009.

[WP02] F.G. Wolff and C. Papachristou. Multiscan-based test compression and hardware decompression using LZ77. In *International Test Conference*, pages 331–339, 2002.