Rekonfigurierbare adaptive Systeme

Gedächtnisprotokoll vom 31. August 2018

Note: 1,3

1 Rekonfigurierbare Systeme

Frage: Was sind rekonfigurierbare (Hardware-) Systeme?
Antwort: Systeme, die zur Laufzeit ihre Eigenschaften verändern können. In Bezug auf Hardware denke ich da an ein FPGA.
Frage: Was für Arten haben wir da kennen gelernt?
Antwort: Grob- und Feingranulare. (Nach langem Bohren wusste ich, welche Antwort er hören wollte...)
Frage: Wie ist ein FPGA aufgebaut?
Antwort: LUTs, CLBs und PSMs erklärt.
Frage: Wie lange dauert eine Rekonfiguration?
Antwort: Je nach Architektur. Bei Feingranularen ein paar Millisekunden, bei den Grobgranularen nur ein paar Zyklen.
Frage: Welche Möglichkeiten gibt es, die rekonfigurierbare Einheit an meine Core CPU anzubinden?
Antwort: Off-Chip Accelerator, Attached, Co-Processor, als RFU on Chip und als Soft/Hard Core in der rekonfigurierbaren Einheit.
Frage: Vor- und Nachteile jeweils?
Antwort: Je weiter weg von der CPU, desto länger dauert der Datentransfer. Man hat keinen Zugriff auf die internen Register der CPU und auch einen höheren Kommunikationsoverhead. Allerdings bedarf es nicht der Neuanfertigung eines Chips, wenn man die Hardware ändern möchte. Man braucht auch nicht die Architektur der Core CPU ändern.

2 Thrashing

Frage: Was versteht man unter Configuration Thrashing?
Frage: Wenn mehrere SIs der Reihe nach in die konfigurierbare Logik konfiguriert werden sollen, aber nicht dafür genug Platz ist. Dann verdrängen diese sich zyklisch und die Ausführung kann länger dauern, als sie es in Software täte.
Frage: Was kann man dagegen tun?
Aufgabe: Man kann zur Compilezeit festlegen, welche Befehle in Hardware gehen,

und welche nicht. *Frage:* Was sind dabei die Nachteile?
Antwort: Wenn sich die Größe des FPGAs ändert, wird sie entweder nicht genutzt (bei größerem FPGA) oder es kommt erneut zu Thrashing (bei kleinerem FPGA).
Frage: Was gibt es noch für Ansätze?
Antwort: Dynamisch zur Laufzeit entscheiden.
Frage: Was muss man hierzu zuzüglich implementieren?
Antwort: Conditional Branch oder Traphandler (auch diese Antwort musste mir aus der Nase gezogen werden).
Frage: Wie funktionrien die jeweils? Was sind Vor- und Nachteile?
Antwort: Conditional Branch ist ein Assemblerbefehl, der einem SI Aufruf vorangestellt sein muss. Dieser prüft, ob die SI in Hardware implementiert ist. Dies hat einen ständigen Overhead. Mit dem Traphandler entsteht nur ein Overhead (der aber größer ist), wenn eine unimplemented SI exception geworfen wird. Dies eignet sich für SIs von denen man zur Compilezeit weiß, dass sie häufig verwendet wird und wahrscheinlich in Hardware verfügbar sein wird.

3 Prefetching

Frage: Was ist Prefetching?
Antwort: Ein Ansatz um die Konfigurationszeiten zu verkürzen. Dabei werden Konfigurationsbits vorgeladen. Direkt in das FPGA, falls Platz ist, andernfalls in den Konfigurationscache, insofern vorhanden.
Frage: Was gab es da für Ansätze? Vor- und Nachteile?
Antwort: Statisch. Welche SIs vorzeitig konfiguriert werden, wird anhand des CFG zur Compilezeit festgelegt. Hierbei entsteht zur Laufzeit kein weiterer Overhead, dafür kann aber nicht auf sich ändernde Abläufe reagiert werden. Dynamisch: ein adaptiver Ansatz. Mit Zählern kann gezählt werden, wie oft welche SI tatsächlich gebraucht wurde und dementsprechend wird ein Prefetching ausgeführt. Hybrid: Manche Befehle werden gemäß Annotation im Quellcode vorzeitig geladen, die anderen werden zur Laufzeit entschieden.
Frage: Was gibt es für Möglichkeiten, dynamisches Prefetching zu realisieren?
Antwort: (Echt nicht sicher, was er wollte, wir kamen dann auf die Prediction Einheit von RISPP zu sprechen) Es gibt mehrere Parameter: $\alpha\gamma\lambda$, die ein Prediction fine tuning erlauben.
Frage: Was macht der Parameter α?
Antwort: Der bestimmt die Stärke der Error-Back Propagation.
Frage: Wenn Sie nun einen Coprocessor haben, der bereits eine SI vorgeladen hat, und diese soll nun ausgeführt werden. Was muss der Coprocessor tun?
Antwort: Da gibt es dann einen Handler, der die Ausführung der SI anstößt. Dann werden die Parameter vom Stack geholt. Die CPU kann angehalten werden, je nach Implementierung.
Frage: Der hat aber noch gar keine Daten. Was muss er noch tun?
Antwort: Der Coprocessor hat Zugriff auf die Register der CPU, die kann er

sich holen.
Frage: Wie kann er die sich holen?
Antwort: (Unsicherheit) eeeeeh (Ich habe das mit dem Parameter-vom-Stack-holen bereits erwähnt, ich wusste nicht dass ihm diese Formuliernung nicht gefiel, daher bohrte er weiter)
Frage: Naja auf dem Coprocessor läuft jetzt der Assembler Code, den sonst die CPU ausführen würde, wie kommt die denn an die Daten?
Antwort: Mit einem Load Befehl. (Das wollte er hören)

4 WARP

Frage: Nun adaptive Prozessoren. Wir haben Warp besprochen. Was gibt es über ihn zu wissen?
Antwort: Loosely-coupled fine grained processor. Bietet eine online Synthese von zur Laufzeit erkannten computational hotspots.
Frage: Wie macht er das?
Antwort: Er hat einen Cache für alle Sprünge, die im Code gelesen wurden. Er protokolliert also die Instruction Fetch phase.
Frage: Und er speichert dann die Sprünge?
Antwort: Ja.
Frage: Nein. Die Adresse des Sprungsziels.
Antwort: Eeh ja. Klar, meinte ich.
Frage: Was gab es da noch für eine Optimierungsmöglichkeit?
Antwort: Das nannte sich auf den Folien Coalescing. Da wurden die Zählereinträge von dem Cache getrennt und nur dort addiert, damit nicht der Cache ständig belästigt werden muss.
Frage: Alle Zähler? Und wie kann ich dann zuordnen, welchen ich inkrementieren muss, wenn ein Sprungbefehl gelesen wurde?
Antwort: Eeeeeh (ich lag falsch, er versuchte mir zu helfen. Langes Ratespiel beginnt... ich kam nicht drauf.)
Bauer: Man speichert nur einen Zähler separat, nämlich den, der zuletzt inkrementiert wurde. Wenn dieser dann nochmal gelesen wird (das ist in rechenintensiven Abschnitten mehrere Tausend mal in Folge der Fall), wird der externe Zähler inkrementiert. Sollte ein anderer Sprungbefehl gelesen werden, wird der separate Zählerstand in den Cache zurückgeschrieben und der aktuelle Sprungbefehl landet im separaten Zähler. Und so weiter.
Frage: Eine letzte Frage noch: Was macht DIM anders, und wieso kann er das, was er macht im Gegensatz zu WARP?
Antwort: DIM erzeugt für jeden Basisblock der gelesen wird eine Ausführung in seiner grobgranularen FU Array. Das kann er direkt zur Laufzeit machen, weil er durch seine grobe Granularität so schnell ist.